PCT/GB 2003 / 002784
10/518991

The Patent Office

INVESTOR IN PEOPLE

The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

PRIORITY DOCUMENT

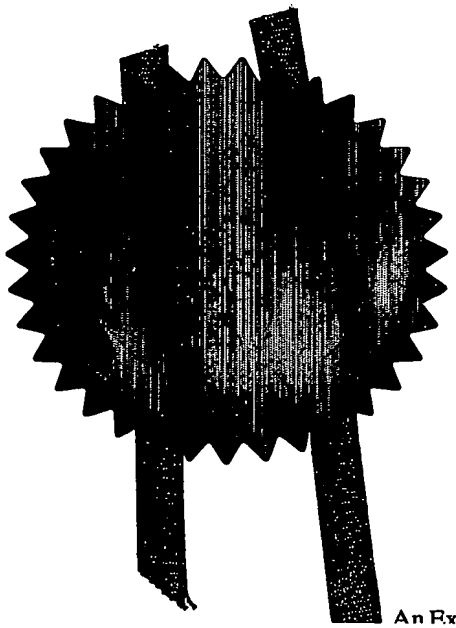
SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.



Signed

Dated 11 July 2003

For Official use only

Your reference **Architecture Gen (UK)**

0215034.0

28 JUN 2002

The **Patent Office**

Request for grant of a Patent

Form 1/77 Patents Act 1977

1 Title of invention

Architecture Generation Method

2. Applicant's details

[X] First or only applicant

2a If applying as a corporate body: Corporate Name

Critical Blue Ltd

Country

GB

2b If applying as an individual or partnership

Surname

Forenames

2c Address

The Scottish Microelectronics Centre
The Kings Buildings
West Mains Road
Edinburgh

UK Postcode EH9 3JF

Country GB

ADP Number 8413175001

☐	Second applicant (if any)
2d	Corporate Name
	Country
2e	Surname
	Forenames
2f	Address
	UK Postcode
	Country
	ADP Number
3	Address for service
	Agent's Name: Origin Limited
	Agent's Address: 52 Muswell Hill Road London
	Agent's postcode: N10 3JR
	Agent's ADP Number: C03274 7270457002

4 Reference Number

Architecture Gen (UK)

5 Claiming an earlier application date

An earlier filing date is claimed:

Yes ☐ No ☒

Number of earlier application or patent number

Filing date

15 (4) (Divisional)	8(3)	12(6)	37(4)
☐	☐	☐	☐

6 Declaration of priority

Country of filing	Priority Application Number	Filing Date

7 Inventorship

The applicant(s) are the sole inventors/joint inventors

Yes [ ] No [X]

8 Checklist

		Continuation sheets	
Claims	0	Description	45
Abstract	0	Drawings	0

Priority Documents	~~Yes~~/No
Translations of Priority Documents	~~Yes~~/No
Patents Form 7/77	~~Yes~~/No
Patents Form 9/77	~~Yes~~/No
Patents Form 10/77	~~Yes~~/No

9 Request

We request the grant of a patent on the basis of this application

Signed: Origin Limited Date: 28 June 2002

(Origin Limited)

Architecture Generation Method

1 Problem Statement

A processor is formed from a number of individual functional units. Each of those units is able to perform different types of computation or storage tasks. In superscalar or VLIW machines there are often duplicates of the most heavily utilised functional units. This allows more parallel computations to be performed. The processor architect determines the replication of and connectivity between these units. This is usually done on the basis of generalized statistics about the likely mix of instructions found in a wide range of real world applications. This means that the processor is designed for a very general workload and the instruction unit mix might not be ideal for any particular application domain.

If an application specific processor is being synthesized then far greater efficiency can be obtained by determining the number and mix of execution on the basis of the particular application domain it is to be used for. The mix must be determined automatically by analyzing the type of code that the processor is going to execute. It is not sufficient to just measure the frequency of usage of different operations and replicate units on that basis. There is no advantage to be gained unless there are situations in which a number of replicated functional units could be usefully used in parallel. The optimisation must perform detailed analysis of the data flows within the program (especially the most performance critical parts) and use that to determine the right number of, and connectivity for, the functional units.

Since the number and type of functional units and their connectivity may vary, so may the instruction word representation. An efficient instruction word representation is required. A processor without a fixed architecture provides a particular challenge in this regard. The average code density that is achieved is very heavily dependent on the efficiency of the instruction word representation. Good code density is an important attribute for an embedded processor where the program needs to be permanently stored in expensive non-volatile semiconductor memory.

2 Prior Art

The nature of existing processor architectures is that the Instruction Set Architecture (ISA) is fixed. This means that the types of operations supported and thus the presence of appropriate functional units is also fixed. The unit mix is determined very early in the design process upon the basis of a wide range of applications. The concept of functional unit selection and replication being integrated into software tools on the basis of application code is commercially novel.

The most relevant academic prior art is that for the automatic synthesis of TTAs (Transport Triggered Architectures). In this approach the starting point is a fully connected TTA with a number of duplicated functional units. The least used connections and least used functional units are gradually removed to produce a more optimised architecture. The performance of the architecture versus its silicon area can be plotted. This is similar to the techniques employed by CriticalBlue. The CriticalBlue approach differs in that connections and resources are added rather than removed. The architecture starts within a minimum number of connections and functional units and new ones are added as required. The CriticalBlue approach has a tighter integration between the code generator and the architectural optimisation. It also allows evaluation of different instruction word widths. The prior art TTA approach uses a fixed instruction word representation and a fixed number of connection buses (with a variable number of connections to those buses).

3 Summary of Contribution

An initial candidate architecture is created with a minimal connectivity network and one copy of each functional unit. Software code is then generated for the candidate architecture. As trial code is being generated, information is collected in a table about resources or connections that could be used to improve the code generation. For instance, a count is kept of the number of occasions in which data needs to be transported between two functional units, and there is not a direct connection between. A count is also maintained of the number of times that all functional units of a particular type are busy with existing operations when another operation of that type needs to be performed.

The counts produced during code generation are then weighted by two different factors. Firstly, code from the software functions that have been marked as being critical for overall performance are given a higher weighting. Instructions within the inner loops of such functions are given a still higher weighting. The weightings are used to bias the resource table. The resource table represents changes to the architecture that would provide the greatest performance benefit to the architecture.

The table is then examined and a single new resource is added. This might be a duplicate functional unit or a new connection between functional units. The area overhead of the new resource is compared against its weight in comparison to other potential resource additions. A choice is made for a new resource taking into account both the area it occupies and its potential benefit. When a new resource has been added to the architecture the code generation process is repeated. The addition of the resource should improve the performance of the architecture and also reveal what further resources should be added.

The instruction word representation is optimised at the same time as the resources within the processor. The width of particular operations increases as new connections are added and thus the number of bits to control the operand multiplexers grows. As code is being generated a frequency table is maintained of how often particular operations are utilised. Again this is weighted by performance critical functions and inner loops so that they have more influence on the architecture. The instruction word layout attempts to allocate different positions in the instruction word for the most frequently used functional units. Thus simultaneous operations on these functional units can be issued without interference. Less frequently used functional units have their control bits overlaid with other functional units in the instruction word. Operations cannot be issued to both simultaneously. Since such a clash happens less often it has a smaller impact on overall performance. The width of the instruction word is varied as part of the architectural optimisation process. The instruction word can be any byte width of 32 bits or wider.

A graph of estimated performance versus area for every candidate architecture is produced. The user is able to pick any architecture that has been produced and fix that as the final architecture for the system. In this way the architecture is molded to the particular requirements of the application.

4 Architectural Overview

4.1 General Philosophy

One of the key requirements of the architecture is to support scalable parallelism. The basic structure of the micro architecture and the operation of the design tools are all focused on that goal.

Extracting parallelism from highly numeric loop kernels is relatively straightforward. Such loops have regular computation and access patterns that are easy to analyse. The nature of the algorithms also tends to lend itself well to parallel computation. The architecture just needs to balance the availability of computational resources (such as adders, multipliers) and memory units to ensure the right degree of parallelism can be extracted. Such numeric kernels are common for DSPs. The loops tend to lack any complex control flow. Thus DSPs tend to be highly efficient at regular computation loops but are very poor at handling code with more complicated control flow.

Other than in numeric computation loops, C and C++ code tends to be filled with complicated control flow structures. This is simply because most control code is filled with conditional statements and short loops. Most C++ code is also filled with references to main memory via pointers. The result is a code stream from which it is extremely difficult to extract useful amounts of parallelism. In average RISC code, approximately 30% of all instructions are memory references and a branch is encountered every 5 instructions.

General purpose processors for PCs have to deal with this kind of code and extract parallelism from it in order to achieve competitive performance. The complexity of PC processors has mushroomed in recent years to try and deal with this issue. The control logic for a modern PC processor has literally millions of transistors dedicated to the task of extracting parallelism from the code being executed. The extra hardware needed to actually perform the operations in parallel is tiny in comparison with the logic required to find and control them. The main method utilised is to support dynamic out-of-order and speculative execution. This allows the processor to execute instructions in a different order from that specified by the program. It can also execute those instructions speculatively, before it knows for sure whether they should be executed at all. This allows parallelism to be extracted across branches. The difficult constraint is that the execution must always produce exactly the same answer as would result if the instructions were executed strictly one by one in the original order.

The control and complexity overheads of dynamic out-of-order execution are far too high for a CriticalBlue processor. There is a significant cost overhead due to the area occupied by the control logic, not to mention the cost of designing it. Additionally, such logic is not amenable to the scalability requirements of the CriticalBlue architecture.

A number of recent developments in the area of micro architecture have been focused on VLIW type architectures. There is a "back to basics" movement that seeks to place the burden of extracting parallelism on the compiler. The compiler is able to perform much greater analysis to seek parallelism in the application. It is also considerably simpler to develop than equivalent control logic. This is because the control logic must find the parallelism as the program is running so must itself be highly pipelined and suffers from the physical constraints of circuit design. The compiler performs all of its work up front in software with the luxury of much longer analysis time. For most classes of static parallelism, compiler analysis is very effective.

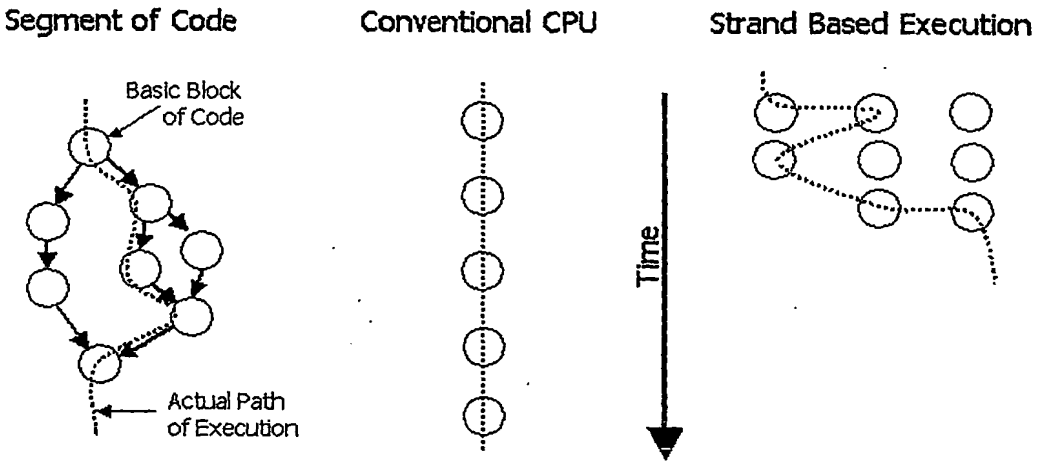
Unfortunately, software analysis is poor at extracting parallelism that can only be determined dynamically. Examples of these are branches and potentially aliased memory accesses. A compiler can know the probability that a particular branch will be taken from profiling information, but it cannot know for sure whether it will be taken on any particular instance. A compiler can also tell from profiling that two memory accesses never seem to access the same memory location, but it cannot prove that will always be the case. Consequently it is not able to move a store operation over a potentially aliased load operation as that might affect the results the program would generated. This restricts the

amount of parallelism that can be extracted statically in comparison to that available dynamically.

CriticalBlue employs a unique combination of static and dynamic parallelism extraction. This gives the architecture access to high degrees of parallelism without the overhead of complex hardware control structures. The software tools perform all the analysis, moving the complexity burden away from the hardware. The CriticalBlue architecture itself is fully static in its execution model. It executes instructions in exactly the order specified by the tools. These instructions may be out of order with respect to the original program, if the tools are able to prove that the re-ordering does not affect the program result. This reordering is called instruction scheduling and is an important optimisation pass for most architectures, and especially for CriticalBlue.

CriticalBlue has a revolutionary execution model that also allows it to perform out-of-order operations that cannot be proved as safe at code generation time. In general it only perform these optimisations if it knows that they will usually be safe at execution time. The hardware is able to detect if the assumptions are wrong and arrange a re-execution of the code that is guaranteed to produce the correct answer in all circumstances. The hardware overhead for this "hazard" detection and re-execution is very small.

The diagram below illustrates the power that this style of execution gives the CriticalBlue architecture:



An example segment of code is shown on the left hand side. This is composed of individual basic blocks. A basic block is a segment of code that is delineated by a branch operation. If execution enters a basic block then all instructions within it will be executed. At the end of the basic block there is a branch instruction that causes execution to continue with one or two different possible successor blocks. The condition upon which the branch is performed is generally calculated in the code of the basic block so it is not possible to know before entering the block which route will be taken. Each execution of the code will produce a particular path of execution through the basic blocks. Certain paths may be considerably more likely than others but any route may be taken.

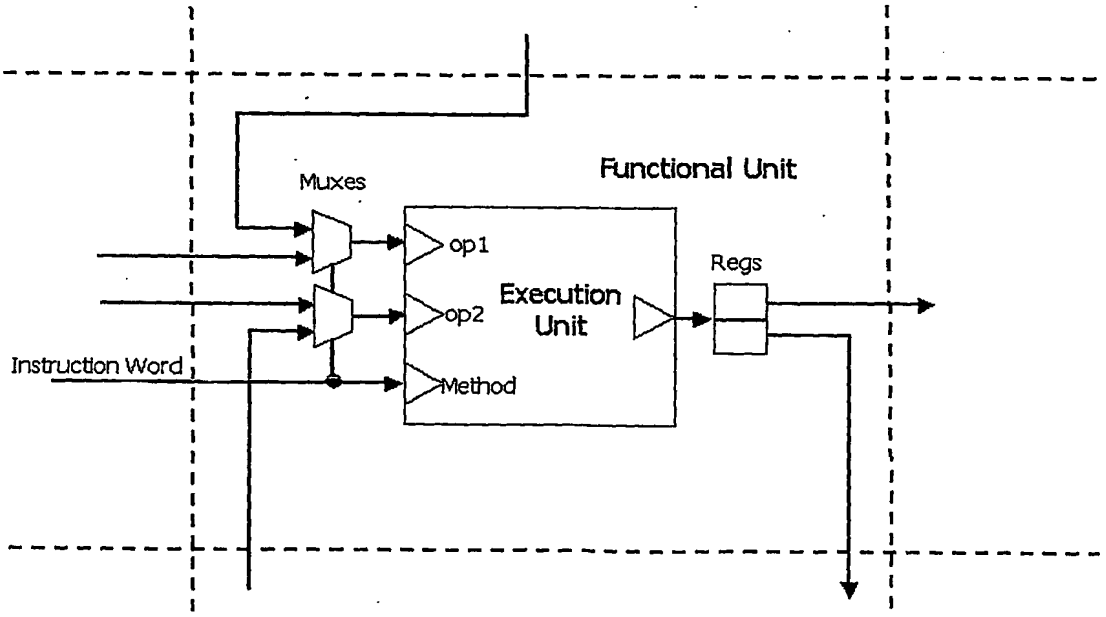
The middle section of the diagram shows the execution in a conventional processor that does not support any kind of out of order execution. This is typical of RISC processor cores. Each basic block as to be executed one after the other, as the branches are resolved.

The right hand section shows the execution model for CriticalBlue. It is able to execute code from a number of different basic blocks in parallel. It does this to increase the amount of parallelism and efficiency of the architecture. It might know that one basic block is very likely to follow another. It can pull instructions forward from the second block to execute in parallel with the instructions of the first. This allows calculations to be started earlier (and thus finish earlier) and to more effectively balance the resource utilisation of the processor. The CriticalBlue scheduling algorithm does this while taking account of the probability that the execution of a particular instruction will be useful.

Executing code speculatively in this manner normally requires a significant hardware overhead. If a particular block should not have been executed then any results it has produced must be discarded. This is referred to as "squashing" the execution. In particular, any store operations that the code has performed must be undone as they could permanently pollute the memory space with incorrect results. CriticalBlue employs mechanisms that allow the benefits of speculative execution while only requiring the minimum of hardware overhead.

4.2 Functional Units

The internal architecture of functional unit is shown below:



The central core of a functional unit is the execution unit itself. It performs the particular operation for the unit. New functional units may be created using user defined execution units. The CriticalBlue tools automatically instantiate the required "glue" blocks around the execution unit in order to form a functional unit. These glue blocks allows the functional unit to connect to other units and to allow the unit to be controlled from the instruction word.

Functional units are placed within a virtual array arrangement. The dashed lines shown on the diagram illustrate this. Individual functional units can only communicate with near

neighbours within this array. This spatial layout prevents the architectural synthesis generating excessively long interconnects between units that would significantly impact clock speed.

The control inputs include the segment of the instruction word that controls that particular functional unit. The method selector is passed directly to it. Other fields are used to control the multiplexers that select data from buses. Each operand input to the execution unit may be chosen from one of a number of potential data buses using a multiplexer. In some circumstances the operand may be fixed to a certain bus, removing the requirement for a multiplexer. The number of selectable sources and the choice of particular source buses are under the control of the CriticalBlue architectural optimisation tools.

All results from an execution unit are held in independent output registers. These drive data on point-to-point buses connected to other functional units. The point-to-point nature of these buses minimises power dissipation and propagation delays. Data is passed from one functional unit to another in this manner. The output register holds the same data until a new operation is performed on the functional unit that explicitly overwrites the register.

4.3 Communication Architecture

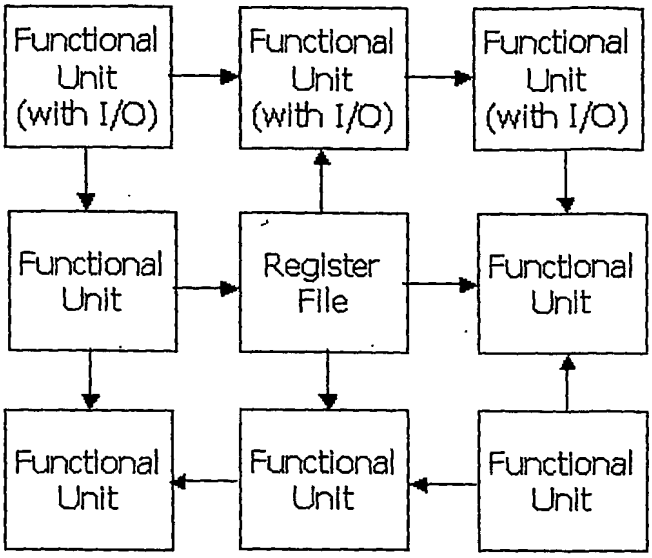
Although the CriticalBlue architecture does have a central register file it is treated like any other implicit functional unit. All accesses to the register file have to be explicitly scheduled as separate operations. Since the register file acts like any other functional unit its bandwidth is limited. The code is constructed so that the majority of data values are communicated directly between functional units without being written to the register file.

Traditional architectures have a centralised register file that has customized access ports to all of the functional units. Access to the register file is implicit in the instruction layout and semantics of the instruction set. The register file is used to feed the operands of the execution units and hold the results generated by them. Unfortunately such a centralised register file imposes a significant restriction on scalability. As the level of parallelism in the instruction stream increases so does the number of access ports required to a centralised register file. These are needed to provide operands to and write back results from all the active execution units. The complexity of the register file grows at approximately N^3 where N is the number of access ports. The register file soon becomes the bottleneck in the design and starts to have a strongly detrimental affect on the maximum clock speed.

Given the requirement to make CriticalBlue highly scalable, communication of all data through a centralised register file is not a viable architectural option. Whenever a functional unit generates a result it is held in an output register until explicitly overwritten by a subsequent operation issued to the unit. During this time the functional unit to which the result is connected may read it.

A single functional unit may have multiple output registers. Each of these is connected to different functional unit or functional unit operand. Since all connection buses are single source/sink there is a one-to-one correspondence between output registers and connections. The output registers that are overwritten by a new result from a functional unit are programmed as part of the instruction word. This allows the functional unit to be utilised even if the value from a particular output register has yet to be used. It would be highly inefficient to leave an entire functional unit idle just to preserve the result latched on its output. In effect each functional unit has a small, dedicated, output register file associated with it to preserve its results.

An example functional unit array is shown in the diagram below:



Given the connectivity limitations of the functional unit array, not every unit is connected to every other. Thus in some circumstances a data item may be generated by one unit and needs to be transported to another unit with which there is no direct connection. The placement of the units and the connections between them is specifically designed to minimise the number of occasions on which this occurs. The interconnection network is optimised for the data flow that is characteristic of the required application code.

To allow the transport of such data items, any functional unit may act as a repeater. That is it may select one of its operands and simply copy it to its output without any modification of the data. Thus a particular value may be transmitted to any operand of a particular unit by using functional units in repeater mode. A number of individual "hops" between functional units may have to be made to reach a particular destination. Moreover, there may be several routes to the same destination. The code generator selects the most appropriate route depending upon other operations being performed in parallel.

There are underlying rules that govern how functional units can be connected together. Local connections are primarily driven by the predominate data flows between the units. Higher level rules ensure that all operands and results in the functional unit array are fully reachable. That is, any result can reach any operand via a path through the array using units as repeaters. These rules ensure that any code sequence involving the functional units can be generated. The performance of the code generated will obviously depend on how well the data flows match the general characteristics of the application. Code that represents a poor match will require much more use of repeating through the array.

CriticalBlue uses a truly distributed register file concept. The main register file does not form a central bottleneck that would severely hamper the scalability of the architecture. The majority of communication in the CriticalBlue architecture occurs directly between functional units without a requirement to use the central register file. Small blocks of registers local to each functional unit hold data values until they are required by subsequent operations. The connections between the functional units and their placement are optimised synergistically to adapt to the data flow present in the application.

5 Hardware Configuration File

5.1 Philosophy

The hardware configuration file describes the attributes of the hardware units in a CriticalBlue processor. All units are listed including control units and both implicit and explicit functional units. Glue units are not explicitly defined but are referenced in the definitions of other units. The hardware configuration file is usable as both an input to and an output from the MetaMapper.

The exact number and type of units within the processor is defined by the contents of the configuration file. However, the type of glue units selected in order to create the required connections is dependent upon the optimisations performed by the MetaMapper. Thus in the input file a choice of glue units, chosen depending upon the number of connections selected, is provided. Each candidate processor generated produces a fixed hardware configuration file. A fixed configuration file has exactly the same format as the input form except that one particular glue unit is chosen whenever a choice of several is made available. The fixed form also shows which connections are made between individual units.

This philosophy of using the same file format for both input to and output from the MetaMapper greatly simplifies the design environment. It allows small incremental changes to be made to the architecture of a CriticalBlue processor. This is especially important if some of the lower level hardware design steps have been completed when the need for a change to the processor architecture becomes apparent. Addition and subtraction of individual connections can easily be achieved by hand modification of the configuration file. The MetaMapper is able to read the modified file for code generation and produce compatible code. The existing connections will be preserved so work undertaken on place and route or achieving timing closure is preserved.

The following sections list the attributes that are detailed for each unit in the configuration file. The ordering of the units in the file is not significant.

5.2 Unit Name

Each unit must have a unique name. For a functional unit this is the name of the execution unit embedded within it. This is used when instantiating the unit in the structural HDL generated by MetaBuilder. Implicit functional units have fixed names starting with the prefix `hw_`. The code generator references these when instructions requiring the unit are translated.

5.3 Number of Ports

Specifies the number of ports associated with the unit. The majority of units only have a single port associated with them. There is an identical set of method selectors, immediate fields, operands and results for every port of a unit. Thus providing multiple ports is equivalent to providing multiple independent. The preference is generally for the architectural synthesis to replicate units as required depending upon the parallelism available within an application. Multiple ports to a single unit are used when all ports must access a single view of internal state. Thus multi-porting is used for memories and register files.

5.4 Decode Position

The decode position is only specified for fixed configuration files. It shows which individual bits in the execution word form the selection opcode for the unit and the control bits for the

unit operands. Control units are not connected to the execution word bus in this manner so do not have decode positions specified.

The MetaMapper is not able to generate new architectures that have a mixture of specified and unspecified decode positions. The execution word optimisation requires that all positions are floating. The decode positions are specified in a fixed file since they are required for code generation. If the user adding additional units to an already fixed file by manual modification of the configuration file then the user must specify suitable decode positions.

The decode position is specified by four different attributes as follows:

5.4.1 First Opcode Bit

This specifies the first bit in the execution word that forms part of the operation code bus for the unit. The value of this bus is compared against a fixed, allocated value, for each unit to determine if it is selected on that clock cycle.

5.4.2 Opcode Width

This specifies the total width of the operation code bus that is distributed to the unit.

5.4.3 Opcode Value

This specifies the value required on the operation code bus to cause the unit to be selected. Unique operation codes (within each distinct operation code bus) are automatically allocated as part of the architecture optimisation process.

5.4.4 First Control Bit

This specifies the first bit in the execution word that forms the control bus for the unit. The control bus contains the method field, any immediate field, immediate strand and the multiplexer settings for each of the input ports to the unit. The width of the control bus is determined by the combined width of the method port, immediate port, immediate strand and input port selection widths.

5.5 Method Selector Port

This specifies the width of the unit method port in bits. A zero length method is port acceptable. Control units do not have a method port and set this field to 0.

In general the method port is used to select which particular method supported by a functional unit is to be performed. The selected method will determine which operands need valid values and which output results will be generated. Each method has a unique method number that is specified in the list of methods for the unit.

5.6 Immediate Port

This specifies the width of any immediate port associated with the unit. If an immediate port is not required then a value of 0 width can be specified. Control units have this field set to 0.

The immediate port is adjacent to the method port and is held in the same latch for presentation to the execution unit at the start of the execute cycle. It is used for specifying immediate values to the execution unit. For instance, for immediate units it is used for specifying the actual literal value. For other types of unit it may be used to specify bit significant values that are used by all methods.

5.7 Immediate Strand Flag

This flag indicates if an immediate strand is specified for a functional unit. All units that can perform operations that can permanently change the machine state must have associated immediate strands. This allows their operation to be disabled if the associated strand is squashed. Units that are only used for performing speculative operations (such as some arithmetic units) do not need to specify an immediate strand value.

Control units have this flag set to false.

5.8 Controller Glue Units

A list of possible controller glue units to be used with the unit is provided. If the no controller should be used or the unit is a control unit then the list is empty. The glue units are specified by the appropriate module names that are instantiated as required by the structural HDL generated by MetaBuilder.

5.9 Delay Pipeline Glue Unit

This field gives the name of the delay pipeline glue unit that should be used. This delay pipeline glue unit is used to delay the strand number for the operation by the same number of cycles as the latency of the execution unit. The pipeline unit is selected using a module name that is used to instantiate it in the structural HDL generated by MetaBuilder. A single pipeline unit is selected to correspond to the latency of the unit.

No pipeline glue unit is specified for control units. If no pipeline glue unit is required then this field is left blank.

5.10 Run-Out Glue Unit

This field gives the name of the run-out glue unit that should be used if any. A single run-out unit is able to handle all result ports from the functional unit. The run-out should be compatible in terms of maximum delay cycles and total width for the functional unit. The run-out is able to hold the state of outputs from the functional unit if there is a pipeline stall or interrupt. The run-out unit is selected using a module name that is used to instantiate it in the structural HDL generated by MetaBuilder.

No run-out glue unit is specified for control units. If no run-out glue unit is required then this field is left blank.

5.11 Input Ports

This is the list of input ports to the execution unit. The list must include all ports even if only some are need to have valid values for particular methods. Method selector and immediate ports do not need to be specified.

5.11.1 Port Name

This is the name of the port. The specified port name corresponds to the name used in the hardware description of the unit. The structural HDL generated by the MetaBuilder uses the name. If the port is dedicated then it has a unique `hw_` prefix.

5.11.2 Port Width

This is the width of the port. Bit widths between 1 and 32 bits are legal.

5.11.3 Operand Glue Units

This is a list of the operand glue units that may be used to connect the port to result buses in the processor. If there are no operand units specified then the port must be directly connected to a result bus. Dedicated ports do not specify a glue unit. A number of different possible glue units may be listed, each proceeded by a number. The number represents the number of sources the following operand unit supports. The number of sources must be two or more. There must only be one selection for each source number value. The MetaMapper optimise selects the appropriate glue unit for the optimised number of sources for each operand.

For a fixed configuration file only a single operand glue unit is annotated.

5.11.4 Fixed Connectivity

This field lists fixed connections to result buses for a particular input port. Connections are specified by use of the unit and output port name. If no glue unit is listed then only one fixed connection may be listed. The number of fixed connections listed must not exceed the maximum number of sources supported by an item in the glue list. The selection values for the fixed connections are always in ascending order of their listing, starting with 0.

The fixed connectivity list allows the user to override the optimiser to provide specific connections in the architecture. This is useful for small scale manual refinement of the architecture. If only one glue unit is specified with the same number of sources as those listed in the fixed list then all connections are fixed.

For a fixed configuration file the number of connections will directly correspond to the number supported by the operand glue unit that has been specified. If necessary the blank entries are inserted for unused selections.

5.12 Output Ports

All output ports generate valid results N cycles after the method is initiated, where N is the latency of the execution unit. However, certain dedicated output ports must produce a valid output after a latency of one clock cycle.

5.12.1 Port Name

This is the name of the port. The specified port name corresponds to the name used in the hardware description of the unit. The structural HDL generated by the MetaBuilder uses the name. If the port is dedicated then it has a `hw_` prefix.

5.12.2 Port Width

This is the width of the port. Bit widths between 1 and 32 bits are legal.

5.12.3 Result Selector Glue Unit

This is the name of the result selector unit to be used for the port. A single result selector must be chosen that is compatible in terms of its width to the output port and the total number of operands. The result selector is used when the unit is being used for transport copying and selects the data present on a particular operand port. The structural HDL generated by MetaBuilder uses the name. Dedicated ports, or output ports from control units, do not need a result selector glue unit so the field is left blank.

5.12.4 Output Bank Glue Unit

This is a list of the output bank glue units that may be used to connect the result port to other input operands in the architecture. Dedicated ports do not specify an output bank.. A

number of different possible output bank units may be listed, each proceeded by a number. The number represents the number of registers the unit supports. There must only be one selection for each value. Each unit must be compatible in terms of the data width supported. The MetaMapper optimise selects the appropriate output bank unit for as required by the number of operands being fed by it.

For a fixed configuration file a single output bank unit may is annotated.

5.13 Latency Attribute

This indicates the fixed latency of the functional unit in terms of clock cycles. All results from the functional unit from all methods must have the same latency. The minimum latency is 1 and the simulation environment fixes a maximum latency. If an operation actually has a variable latency then the most frequently occurring fixed latency should be specified. This may be extended dynamically using the pipeline wait mechanism.

High latency execution units can lower code density when they are used as the region must be at least as long as the highest latency of unit used in the region. Code may have to be padded with no-operations while the results are being calculated in the unit.

5.14 Method List

A list of the methods available for the unit must be specified. A control unit does not require any methods to be listed, as it is implicit that it performs a fixed operation on each clock cycle. All functional units must specify one or more methods. If a the unit has a zero length method selector field (and therefore only one method) then that single method must be listed with a method number of 0.

Each method may read input ports and write to output ports. A parameter list is specified for each method to show which subset of the ports for the unit is read/written. These correspond directly to the parameter list for a software function call.

The following are the fields that must be specified for each individual method:

5.14.1 Method Name

This is the name of the method and corresponds directly to a function identifier. Thus all method names must start with the `hw_` prefix. All calls to the function of the given name are automatically converted into a use of the hardware method. All method names must be unique, both within a particular unit definition, and globally across all unit methods. Some method names for implicit functional units are predefined.

5.14.2 Method Number

Gives the number of the method. This is the value that is presented into the method selector field to initiate that particular method. Decode logic within the execution unit must select the corresponding operation. All method numbers for the unit must be unique and small enough to be specified in within the width of the method selector port.

5.14.3 Return Value

Gives the name of an output port to be used for a return value. This must be the name of an output port for the unit. If the software function for the method returns no result (i.e. it is a `void` function) then this field is left blank.

5.14.4 Parameters

Specifies the port names of the parameters to the method. There is a one-to-one correspondence between these parameters and the software parameters of the

corresponding function. Each port name can be either an input or output port. Input ports correspond to pass-by-value (i.e. parameters to) the function. Output ports correspond to pass-by-reference parameters to be used for passing results back from a function. The parameter passed to an output port in a function call is taken to be the address of where the result should be written. Code is automatically generated to read the value from the output port and write it to the specified memory location. Support of output ports in the parameter list allows methods to return multiple results.

Any parameter may be marked as DISCARD. Any discarded parameter is not passed to or written back from the hardware unit. This allows use of functions that pass redundant parameters that have no effect on the results returned from the hardware unit. Such parameters may be supplied for consistency or for future expansion. In particular, the parameter is useful for discarding the initial this pointer from functions that are non-static member functions of C++ classes. The this pointer has no use for hardware implementations of such methods.

5.14.5 Speculation Attribute

This is a flag that indicates if the method may be issued speculatively. Such a method must not result in a side effect that could affect the correctness of program results if the method invocation is discarded. A speculatively issued method may be re-issued any number of times if the region is restarted.

In general as many methods as possible should be marked as suitable for speculation. This gives the CriticalBlue tools the maximum freedom in the scheduling of code and maximums utilisation of the processor pipeline.

Methods that may not be speculated include stores, squash operations and any operation that generates an output external to the functional unit.

5.14.6 Blockage Attribute

This indicates the number of clock cycles before another method may be issued to the functional unit. Some methods may cause a blockage if the implementation of the execution unit is not fully pipelined.

The blockage value may be between 1 and a maximum value imposed by the simulation system. If a method has a blockage of 1 then that indicates that another method may be issued to the same functional unit on the following cycle.

The blockage value may actually be set to be greater than the latency of the unit. This facility is provided for execution units that might have to perform internal "housekeeping" functions before being able to accept a new operation.

High blockage methods can lower code density as at least the blockage number of cycles must be left from the point of issue of the method to the end of the region. This ensures that the next region can use issue a method to the unit on its first cycle without violating the blockage. Code may have to be padded with no-operations at the end of the region in order to achieve this.

5.14.7 Writing Dependence Set

This field lists the dependence sets to which the method writes. A dependence set is simply defined by using a unique identifier. If a method writes to a dependence set then it must maintain its original issue position with respect to all other writes to the same set. This ensures that semantics are correctly preserved for methods that have side effects that modify internal state within a functional unit. Writing methods also form sentinels over

which reading methods from the same set cannot be moved. The dependence sets are global across all functional units so ordering can be maintained between operations issued to different units. A particular method may write to any number of dependence sets (including none).

5.14.8 Reading Dependence Set

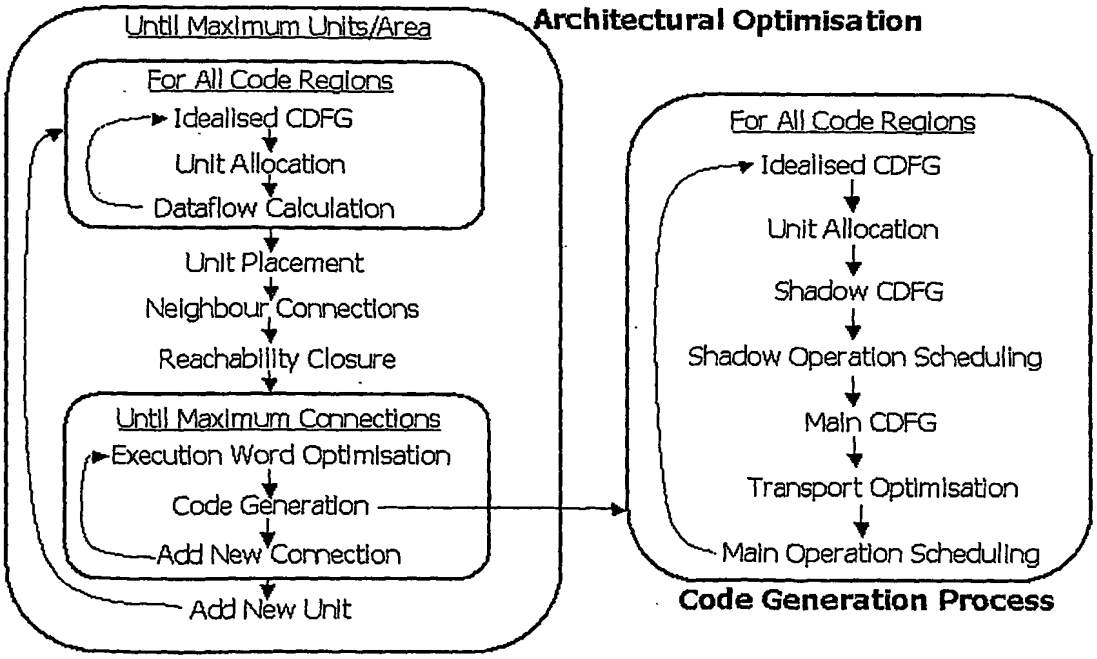
This field lists the dependence sets from which the method reads. This restricts movement of these methods across writes to the same dependence set. Reads from a particular set may be freely scheduled with respect to each other. However, a read from a dependence set may not be scheduled over a write to the same set. A particular method may read from any number of dependence sets (including none).

As an example, dependence sets may be used for a floating point unit that includes a sticky overflow bit. The sticky bit represents internal state that impacts the order in which floating point operations may be performed. Assume there is a sticky bit read operation that obtains the state of the sticky bit and then clears it. This will be represented as writing to the floating point dependence set. All the operations (such as add, multiply, subtract etc) that may set the sticky bit will be represented as readings of the dependence set. This is because the order of the arithmetic operations will not impact the final state of the sticky bit (any overflow causes it to be set). However, arithmetic operations must not be moved over the clear and read of the bit.

6 Optimisation Flow

6.1 Flow Diagram

The diagram below shows the flow of individual optimisation steps for both architectural optimisation and the code generation process:



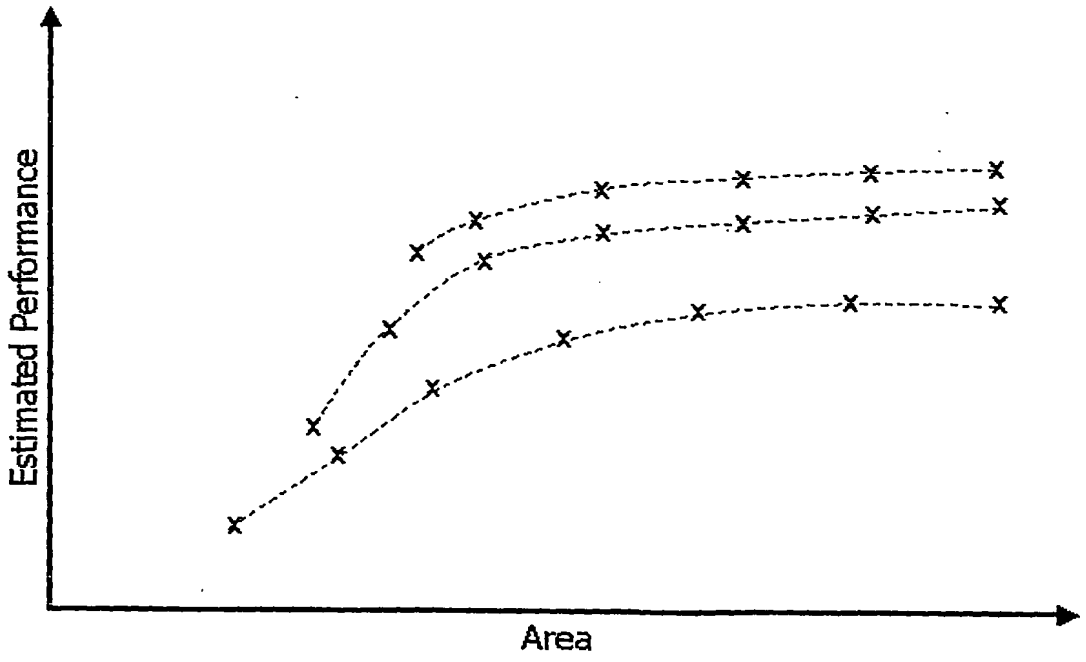
Once the architecture has been fixed and new code is to be targeted to a processor then only the code generation process needs to be performed. As shown the code generation process is actually a subset of the full architectural optimisation.

During the architectural optimisation an initial architecture is created with minimal functional unit and interconnect resources. Additional interconnect is then added to improve the performance of the architecture. This is measured by generating code schedules on the candidate architecture and estimating their performance. When a certain threshold of new connection addition is reached then additional functional units may be added to improve parallelism. Whenever new functional units are added the interconnects are restored to the initial minimal connectivity. The connectivity optimisation process is then repeated.

6.2 Architecture Optimisation Output

The output of the architectural optimisation process is a series of candidate architectures with particular performance characteristics. The configuration information for each of the architectures is stored and the user can then select any individual architecture that most closely matches the required area and performance requirements of the application.

The output of the tool is in the form of a graph representation. An idealised representation of the graph is shown in the following diagram:



The graph shows the estimated performance versus area requirements for all the candidate architectures tested. Each cross on the diagram represents such a candidate architecture. A dashed line in the diagram joins all the candidates that are constructed from the same set of functional units. Each candidate has a different number of additional connections added.

In general the shape of the curves produced will be similar to those on the diagram. An initial architecture can be improved by the addition of new connections but ultimately there is a law of diminishing returns as new connections are added. In reality the curves produced will not be nearly as smooth as particular idiosyncrasies of particular architectural configurations will produce a noise level on the general improvement of the architecture as new connections are added.

As illustrated on the diagram the disadvantage of architectures with greater numbers of functional units is that the minimal area becomes larger. In general, however, the average performance should be better presuming that the parallelism afforded by the additional functional units can be efficiently utilised.

The area is estimated by the addition of the area of all the functional units employed. The library definition for each unit has an attribute of the amount of area occupied for a particular target technology. Additional area is also estimated for the connections between the functional units. Heuristics are used depending upon the target technology and the number and width of connections.

The estimated performance is based on a weighted representation of the number of clock cycles to implement the critical functions within the application. The number of clock cycles is obtained by generating code for the critical functions to run on the candidate architecture. The library blocks and connectivity rules are constructed so that the architecture is designed to run at a particular clock frequency for a particular target technology. Thus the clock speed multiplier is designed to be a constant for all candidate architectures generated. If the user selects a particular candidate architecture then additional information is available showing the performance on individual critical functions within the application.

6.3 Architectural Optimisation

The architectural optimisation process occurs in three main phases. These three phases are repeated in order to generate a range of candidate architectures for the user. The optimisation initially starts with the minimum number of functional units. This is one of each type of functional unit required from the library. The quantity of a particular type of functional units is then increased in order to improve the amount of parallelism that can be exploited in the architecture.

6.3.1 Quantitive Dataflow Estimation

The purpose of the quantative dataflow estimation is to generate a table of the data flow between all the functional units in the architecture. Initially the functional units will be the minimal set but will be expanded as additional functional units are added to the candidate architecture. A translation of the source instructions and creation of a CDFG is required for this phase in order to measure the data flow. The quantative data flow estimation allows decisions about the initial placement of the functional units and the basic connections between them to be made.

All the individual steps below are repeated for all the regions in the application program in order to provide data for the whole application. The data flow results from each region are weighted according to the criticality of the functions (and level of loop nesting) from which they come.

6.3.1.1 Idealised CDFG

The first step is to produce an idealised CDFG for the region. Such a CDFG ignores the creation of register file read and write operations to maintain correct state within the registers. This allows the actual data flow between operations to be measured. This step is identical to that performed during the standard code generation process.

6.3.1.2 Unit Allocation

Once the idealised CDFG has been constructed the particular units to be used for operations in the CDFG can be selected. Obviously this is straightforward for cases where there is only a single unit that can perform an operation. If there are multiple units then spatial analysis is used to select an appropriate candidate. This step is similar to that

performed during the standard code generation process. However, since the physical placement of the functional units has not been determined at this stage the allocation does not make any account of the transport costs associated with a particular allocation. The decision is based purely on maximising theoretical parallelism.

6.3.1.3 Dataflow Calculation

This step analyses the idealised CDFG as annotated with the unit allocations. Each of the arcs in the CDFG is examined and its impact is recorded in a dataflow matrix. Entries are weighted depending upon the criticality of the function from which the region is obtained and the loop nesting level. The cummulative matrix produced after visiting all regions represents the overall flow of data between functional units within the system. This step also maintains a frequency of use count for each of the functional units in the system. This is used to order their placement in the processor array.

6.3.2 Base Architecture Creation

Once the dataflow information has been gathered the base architecture can be created. The base architecture is the set of functional units included in the architecture plus the initial connections between them. Further connections are then added to the base architecture to further improve performance. A number of candidate base architectures may be generated as additional functional unit resources are added.

6.3.2.1 Unit Placement

The first step in the base architecture creation is the unit placement. Functional units are placed within a virtual array. The register file is always placed at the centre of the array since it is a key resource and its connectivity requirements will be underestimated due to the use of idealised CDFGs. Placing it at the centre of the array will give it the maximum possible connectivity to other functional units in the architecture. The other functional units are then placed so that a rectangular area of allocated units is maintained. The exact position of the placements is dependent on the dataflows identified during the quantative dataflow estimation. Units are placed in order of decreasing frequency of use so that the most used units are near the centre of the array.

6.3.2.2 Neighbour Connections

Connections are generated between neighbour functional units within the array. The connections that are generated and their direction is determined by the quantative dataflow matrix that shows the amount of communication between particular functional units. Connections are created that reflect the likely predominate data flows through the processor array.

6.3.2.3 Reachability Closure

This step introduces additional connections into the array in order to ensure that all parts of the array are reachable. This means that a result from any functional unit can be transported to the operand of any other unit within the array.

6.3.3 Connectivity Optimisation

The purpose of the connectivity optimisation is to add additional connections to the base architecture improve performance. Each iteration produces a new candidate architecture. New connections are added until a maximum count of new connections is reached. This count is based on the number of functional units in the array. More connections may be added to arrays with more units. This provides new connections being added to the point that the area occupied by the connections becomes much greater than the area occupied by the functional units themselves.

The connectivity optimisation consists of a number of individual steps as described below:

6.3.3.1 Execution Word Optimisation

This step generates the execution word layout for the processor. The execution word width is a fixed input parameter to the optimisation process. The execution word has to be updated after the addition of each new connection has the addition of a new connection may increase the width of a selection field for a particular functional unit. Thus the layout needs to be updated. The layout optimisation uses the frequency of unit use information gathered as part of the code generation process.

6.3.3.2 Code Generation

This step performs the code generation as described previously. This process is identical to that performed when the architecture has been fixed. In this case it is performed on the candidate architecture. All of the application code is analysed as part of the code generation process. However, the microcode does not need to be physically output into a file. The output is the estimation of the performance of the candidate architecture.

6.3.3.3 New Connection Addition

The choice of new connections is driven by the transport optimisation phase of the code generation. This generates a connectivity request matrix for the whole application. The requested connection with the highest weight that can be added is included into the system. There are a number of restrictions on the addition of new connections. Firstly, there is a maximum input multiplexer selection number and a maximum number of output connections from a unit. These are parameterised as part of the library definition so that a fixed target operating frequency can be achieved. If a particular unit has less timing slack the the choice of connections is made more limited. Secondly, there is a restriction on the distance of a connection in the array. Only connections between neighbour and near neighbour units are permitted. The extent of this area can be parameterised as part of the library definition.

6.3.4 New Unit Addition

The new unit addition is the final step in the architectural optimisation process. It is performed when the addition of new connections to a base architecture has been exhausted. A particular type of unit is selected an additional functional unit of that type is made available in the architecture. The choice of the unit type is based on statistics gathered during the code generation process showing the most frequently used units that can be replicated (memory units, register file units, I/O and some other types of units cannot be replicated). The usage count of a unit includes the copy operations required to copy data into and out of the unit. Thus if performance can be increased by adding more units so that they are likely to be more locally available then this is appropriately reflected in the usage count.

6.4 Function Weighting

6.4.1 Philosophy

The critical function list indicates to the architecture optimiser which functions are key to the overall performance of an application. The critical list may be formed on an ad hoc basis by the designer or, more commonly, from performance profiling of the application.

The critical function list allows the optimiser to balance connectivity resources depending upon need to individual blocks of code. If a new connection greatly improves performance of an inner loop in a critical function then this is likely to be a good addition. If however, the relevant code is form an infrequently executed function, then the cost of the addition connection is unlikely to be justified.

6.4.2 Format

A simple format is employed for the critical function list. Each line of the file corresponds to a particular function. The line specifies the name of the function and the weight it should be given in the optimisation process. The weight is specified as a percentage so values between 1 and 100 are acceptable.

The total of all weights in the file should be 100 or less. If the total weight is less than 100 then the remaining percentage is allocated equally amongst all other functions not listed in the file.

Functions may only be listed once in the file and the function must be implemented in software running on a CriticalBlue processor. Functions that are implemented in hardware or in software on the main processor cannot be listed.

7 Architecture Creation

7.1 Overview

The tools must create connections between execution units in response to the data flow in the application. The network must be a good fit for the application but still retain sufficient connectivity so that the processor is still able to execute general purpose code without crippling inefficiency. The optimisation problem may be further constrained by total area requirements or a limit on the number of inputs or outputs for a single operand multiplexer. The goal of the optimisation is to produce an architecture that is minimal, but sufficient. The overall utilisation of the connections and execution units in the application must be very high. Unnecessary connections just add a cost burden to the processor.

There is a strong synergy between the code scheduling and architectural synthesis optimisations. These two problems should not be considered in isolation. If the architectural synthesis produces a good architecture it is of no use if the code scheduler is unable to make efficient use of it. Equally, there is no point in producing good schedules for a fundamentally bad architecture. The CriticalBlue approach intimately ties these two optimisation problems together to avoid these types of mismatches. Indeed, it is the code scheduler itself that decides when to add new connections to the architecture.

The basic approach employed for is to generate trial schedules on candidate architectures. An initial trial architecture is produced with the functional units as specified by the user. A minimal connectivity network between functional units is also generated. This is a minimal architecture that will be optimised by adding customised connections.

The effectiveness of a particular architecture is measured using attributes of the scheduled code. These include the estimated execution time for particular code blocks and average utilization of processor resources. The user supplies a list of functions that are critical to the overall performance of the application. A weight can be annotated for each function to show its contribution to the overall execution time. The functions can be

identified and proportions determined by using profiling tools. For a particular trial architecture the overall architectural fitness calculation is then made on the basis of generated code efficiency weighted by its importance to overall application performance.

When the code scheduler is operating during architectural optimisation it maintains information about new connections that might be added. When trying to transport data between operations it tries to use the most direct routes. If those connections do not exist in the architecture then it adds a request to add them. It then looks at less favourable routes, that may take longer, until it finds a usable route. At each stage requests are made to add new connections. These connection requests are driven directly by the flow of data in the user's application. The weighting of the request is dependent upon the criticality of the node and the importance of the function to which it belongs.

The weighting system automatically concentrates the optimisations onto the most important data paths in the application. The approach is somewhat akin to simulated annealing. The most critical (i.e. hotter) operations can have the biggest impact on the connection requests while the least critical (i.e. cooler) operations have less of an impact.

When the code scheduling for the application is complete the connection requests are examined. The most requested connection may then be added to the architecture. There are restrictions on the maximum fan-in and fan-out for a particular operand or result port for a functional unit. Once the limit has been reached no new connections may be added. In that case the second most requested connection is added, and so on. Once a new connection is actually made, all the code is scheduled again. The existence of the connection should improve the code sequences produced.

Information is also generated showing the utilisation of particular functional units, both on an overall basis and within particular functions. The user can then assess whether there would be a benefit to adding another copy of a particular functional unit. If there are multiple versions of a particular functional unit then the code scheduler automatically balances code across them.

As new connections are added the architecture is being specialised to the application. Fused units, corresponding to a group of individual functional units in a fixed pipeline, can become an emergent property of the architectural synthesis. What is produced is a hybrid of a general purpose and specialised connection architecture, as directed by the function weightings.

Information about each trial architecture is produced for the user. This includes an estimate of its area, its code density and estimates of its performance on particular functions and overall. The user can then choose any of the trial architectures for actually synthesis into a real processor. Thus the user can trade off area and performance as required for their particular application.

7.2 Interconnect Philosophy

One of the key goals of the architectural synthesis is to optimise the interconnections between the functional units. A traditional architecture fully connects all the functional units in the processor via the central register file and various bypass buses. As discussed previously, this does not provide a suitably scalable architecture for CriticalBlue processors.

The connections between individual functional units are customized depending upon the characteristics of the data flow for a particular application. If the results of one functional unit are never, or only rarely, used by another unit then there is little value in having a

direct connection between them. The number of possible connections grows as a square of the number of functional units and each one occupies area on the chip. Also, as the number of connections to a particular operand grows the input multiplexer becomes slower, lowering the potential clock speed for the system.

Each functional unit has a number of operand inputs and outputs. The output values are held in output registers. Each output drives a single source and single sink bus that is connected to an input of one other functional unit.

7.3 Unit Placement

The units in the processor are placed in a virtual array. This phase only places the units, while connections between them are added during the following phases.

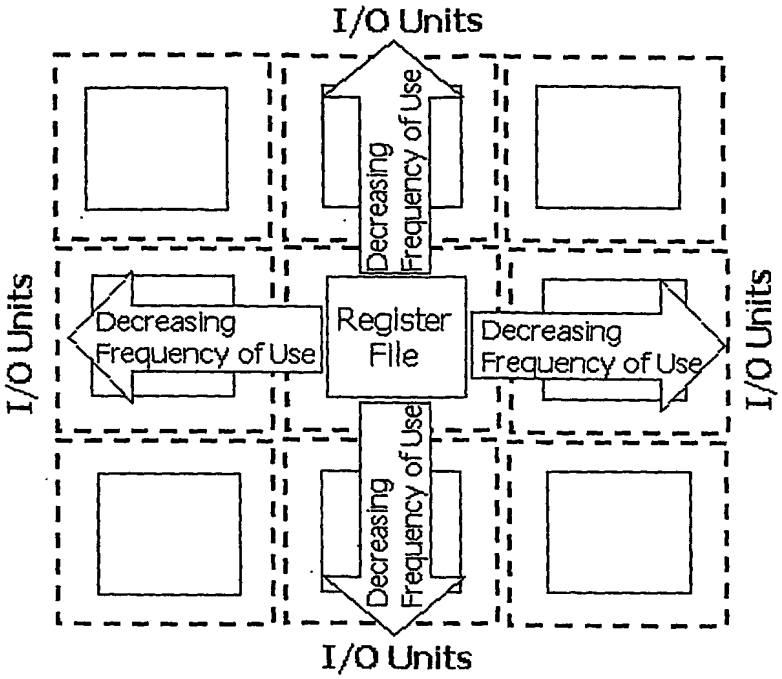
The placement occurs in a virtual grid with the first unit placed in the middle of the grid. The grid can expand in any direction but is forced into a rectangular shape. New units are always added at the boundary with the rule that the maximum extend rectangle cannot be more than one unit larger in any direction than the enclosed fully utilised rectangle. The aspect ratio of the rectangle is a parameter and is used to modify the fitness metric for particular placements to try and force the growth into the required shape.

Placement is driven from an idealized representation of the code. In this form it is assumed that all execution units are able to communicate directly. Thus there are no transport latencies and the actual scheduling is not performed. A weighted data flow matrix is built up as a result of the analysis showing the level of communication between particular execution units. Each row/column represents a particular execution unit operand or result – if there are multiple copies of a unit then they are represented separately.

Placement always starts with the register file. This is chosen because it is a key unit and the amount of communication it requires with other units will be under-represented due to the idealised assumptions made during the CDFG optimisations. Thus placing the unit at the origin increases the opportunities to add additional connections to it later in the connectivity optimisation. The register file is also guaranteed to use 32 bit values, the maximum possible. This allows a simpler algorithm to ensure the reachability of data values of a certain width throughout the unit array.

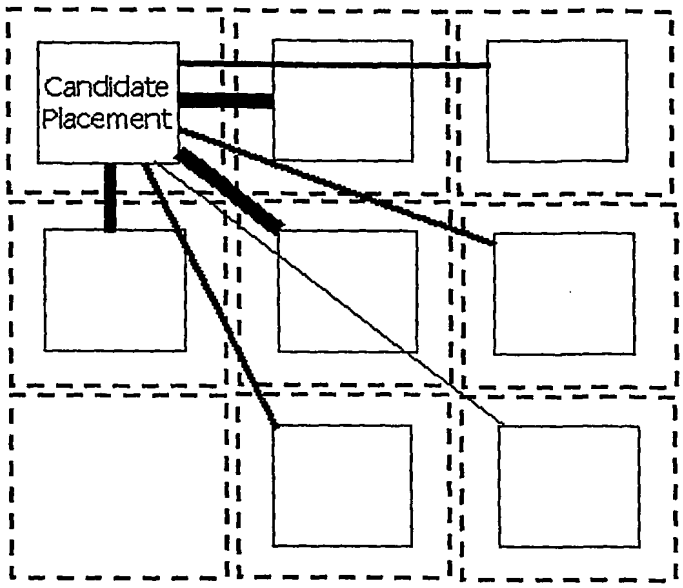
The remaining functional units are placed on the basis of decreasing weighted frequency of use. Thus the placement aims to place the most frequently used units near the centre of the virtual array and the less frequently used units on the periphery. Some units may be marked as being I/O units that must be allocated to the boundary of the virtual array. A particular side of the array may be selected. These units are always placed last independently of their usage frequency. All units that only have inputs or only have outputs are also placed at the edge of the array. These are units are placed on the edge because they are unable to act as repeaters. Note that there may be more units that need to be placed on the boundary than there are boundary positions. In that case the placement does not complete. Additional non-boundary units are replicated until the boundary size is sufficiently big.

The diagram below shows the expansion of the unit placement on the basis of decreasing frequency of use. Certain I/O units are placed on particular boundaries as shown:



The actual positioning a placement is dependent on the communication requirements of the unit. Each free slot on the boundary of the virtual array is considered as a candidate. The communication fitness is calculated by the total amount of communication to each another unit multiplied by the Euclidian distance to the unit in the array. Communication with units of the same type is not included in the calculation (thus reducing the chances of units of the same type being placed close to one another). Thus new units will tend to be placed close to units with which they have high communication. This also means that there is a higher chance that a short cut connection could be added during the connectivity optimisation stage. Obviously this is an approximation since the actual transport cost will depend on the neighbour connections actually created.

The diagram below illustrates the effect of the unit placement Consider that the unit in the op left corner is being placed into the array and that position is a candidate for its final placement. The placement is good as the majority of communication (as shown by the thickness of lines of the diagram) is with neighbour units. Thus direct connections can be created with them avoiding the requirement for transport copies via other units. Other placements of the unit on the other side of the array would be significantly less optimal.



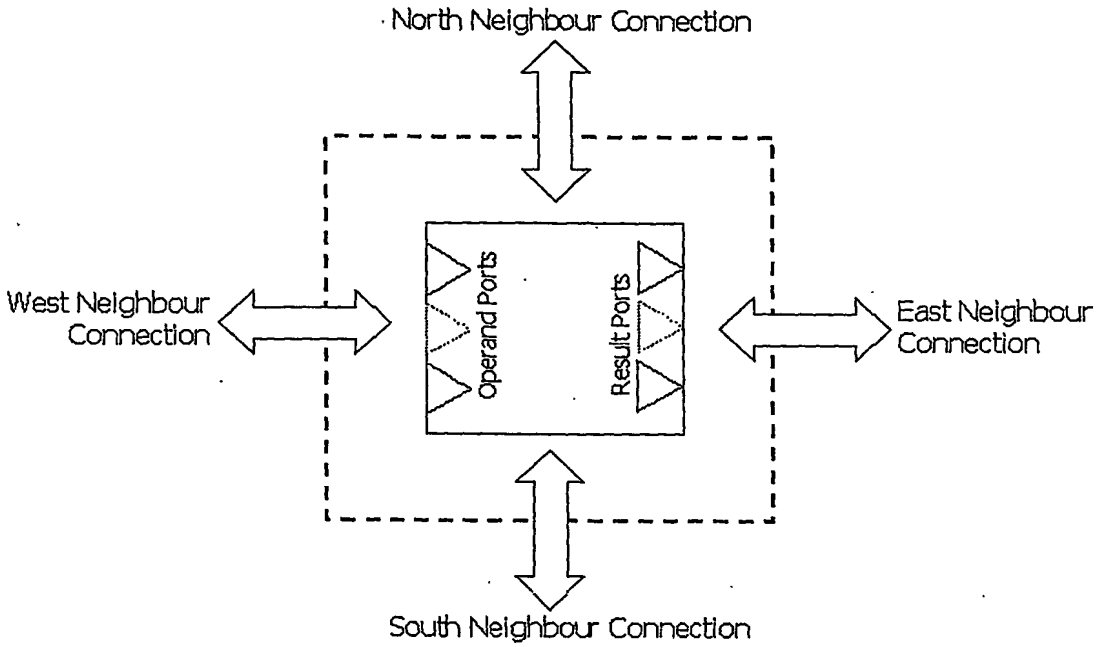
A further restriction controls the placement of the units. This is the requirement to transmit data values without over-truncation to other units in the array. The maximum bit width requirement is maintained for all unplaced units. This is the maximum width of any operand or result port. There must always be at least one position held free adjacent to a unit with an operand and result port the same width or wider than the maximum width of unplaced units. If this prevents any placements being made then the highest unplaced unit may added to the pool of candidates. This mechanism ensures that it is always possible to build routes to transmit data values of the required size to all units in the array.

7.4 Neighbour Connections

This phase adds connections between unit neighbours in the virtual array. All horizontal and vertical neighbours are linked via north-south and west-east connections. This connectivity is mandatory and serves two main purposes. Firstly, it ensures the units are formed into an array that will direct the place and route of the eventual HDL so that it reflects the spatial model created by the tool. Secondly, it forms the basis of the minimum connectivity network for the processor. All units are able to copy operands so if a value is available at any unit in the array it can be transported to any other.

The units are visited in the same order that they were placed in the array (the ordering is stored). A connection is added for each NESW neighbour of the unit. The number of neighbours will vary between 1 and 4 depending upon the unit. The sides are visited in order of distance from the central unit so that those connections (i.e. connections to the centre of the array) are given priority. If a neighbour connection already exists then and additional one is not added. For each unconnected neighbour the most appropriate operand or result port is selected. Multiple connections to particular result ports or operands are not made unless absolutely required (i.e. if a unit only has one operand and result but four neighbours). Neighbourhood links are the width of the ports involved. The arrangement of the units ensures that data values of sufficient width can be passed around the network. If a result port is wider than the operand port it is connected to then superfluous bits are unconnected. If the operand port is wider than the result port then either sign or unsigned extension logic is added depending upon the type of the operand.

The possible neighbour connections that may be made to a unit are illustrated in the diagram below:



The fitness of the connection is formed on the basis of the weighted sum of communications likely to be flowing in that direction based on the data flow matrix. Again this is based on the approximation of Euclidian distance to the required unit.

Once the neighbour connections have been made all adjacent units are connected NESW by one connection. The direction of the connection will depend upon the particular data flows. However, these connections do not guarantee that a particular data value can be transmitted anywhere in the array. The next phase augments the connections to ensure that is possible.

7.5 Reachability Closure

This phase adds additional neighbour connections as required to ensure that any result from any unit can be passed to any operand of any unit. The neighbourhood connections added during the previous phase will provide most of the connectivity required to achieve this. The levels of data flow direct the connections added during that phase. They do not guarantee full reachability. There may be areas of the array that form islands that are not connected to other areas. Moreover, there may be operands and result ports that remain unconnected. This will definitely be the case if there are units with more operand ports and results than neighbours.

A data structure is created with an entry for each unit in the array. Each operand and result port for the unit has an associated flag. For an operand port that flag is set to true if data can be passed to the operand port from the first result port of the origin unit (the unit in the centre of the virtual array). For a result port that flag is set to true if data from it can be passed to the first operand of the origin unit. These flags are used to determine if the array is fully reachable – all flags will be set to true in that case.

The analysis starts with the origin unit and the physical neighbour connections are traversed to set the flags appropriately. Flows into the origin are followed backwards for the result flags and flow out of the origin are followed for the operand flags. The resulting flags represent the initial origin reachability of the units. The reachability analysis

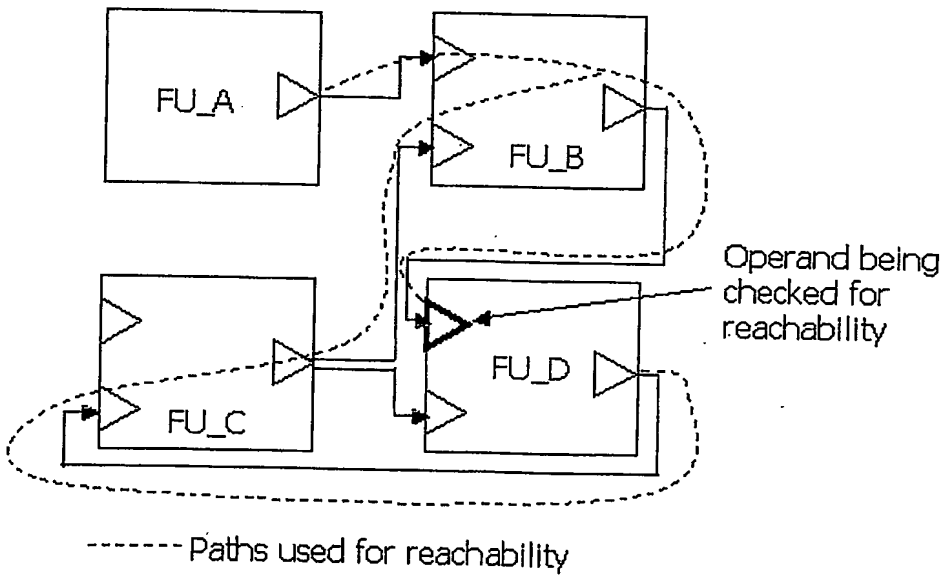
maintains the minimum width connection that has been passed through. Units with operands of greater widths are not considered to be reachable even if connections exist since some of the upper bits of a data value will have been lost.

The reachability analysis does not allow the flow of data through units that already have one or more operands considered to be reachable. All operands must be reachable using routes not requiring use of other operands to a unit. This ensures that it is possible to route all operands to a unit one-by-one and hold them in the connecting bus as live until the unit is triggered. There may also be certain other operands that are exempt from the reachability copy flow because they are required to be live for certain code generation sequences.

The first placed unit for which all flags are not set is adapted. A new connection is added to the first unreachable port to a neighbour unit that is reachable. If no neighbours are reachable then another unit is tried – at least one unit must a reachable neighbour. Each result flag also has an attribute of the number of bits width that are actually reachable so that suitable connections are made. Once the connection is added the analysis is extended to determine which units are now reachable. This process continues until all ports are reachable. This also ensures that all operand and result ports are connected.

The reachability analysis may include a maximum hop distance (not taking account of unit latencies) to provide a ceiling to communication times. This could be set to maximum length of a side of the virtual array rectangle.

The diagram below shows an example of reachability closure analysis for a particular operand:



The dotted lines show the paths that are examined to prove the reachabillity of the operand from all possible result ports. In this case the operand is fully reachable.

7.5.1 Exempt Connections

A number of connections to particular operands on particular units are exempt from usage for transporting data items when determining reachability closure. This is because these operands are required to hold live values across the usage of other operations. The type

and extend of these exempt connections is dependent on the nature of the source instruction set being translated from.

The exempt connections are as follows:

- **Memory Unit Address Operand:** The address needs to be held while performing a shift in order to implement a subword write. The address must be calculated before the shift (as the shift amount is dependent on the address).
- **Memory Unit Store Operand:** The store data needs special storage to allow the implementation of a swap function. The data to be stored is held while a load/store is performed on the other memory location.
- **Shifter Amount Operand:** The shift amount needs to be held while a load operation is performed. A shift may be performed after the load in order to implement a subword load.

7.6 Connectivity Optimisation

The purpose of the connectivity optimisation is to add additional connections to the base virtual processor array to improve performance. This is driven by the transport optimisation phase of the code generation. This generates a connectivity request matrix for the whole application. The requested connection with the highest weight that can be added is included into the system. There are a number of restrictions on the addition of new connections. Firstly, there is a maximum input multiplexer selection number and a maximum number of output connections from a unit. This may be parameterised depending upon the timing slack within the unit itself. Secondly, there is a restriction on the distance of a connection in the virtual array. A circle around the input to the unit represents this. The diameter of the circle is related to the timing slack of the consuming unit.

8 Architectural Synthesis Example

8.1 Introduction

This chapter provides an example of architectural synthesis and code generation. A base architecture is generated with the required execution units. The architecture is then optimised using a performance critical loop of code. Custom connections are added and the impact on the resultant code schedule is shown. The final customized architecture along with the code schedule for the performance critical loop is shown at the end of the example.

This example should help with a general understanding of how the process works in practice and the function of the individual steps.

8.2 Source of Program

The following diagram shows the source code for the code that is going to be used as the example:

```
int sample[1024];        // input samples
int output[1024];        // filtered output samples
int coeff[16];           // filter coefficients
int i, *j, *k, sum;      // temporary variables
...
for (i = 0; i < 1024; i++) {
  sum = 0;
  for (j = &sample[i], k = &coeff[0]; k < &coeff[16]; j++; k++) {
    sum += (*j) * (*k);
  }
  output[i] = sum >> 16;   // shift to take account of fixed point
}
```

The code shows a 16-tap Finite Impulse Response (FIR) filter coded in the C language. This type of digital filter is very common and used in wide range of applications. The central loop multiplies a sample with a coefficient and keeps a cumulative total for the results with all coefficients.

The source has been coded to minimise the number of instructions generated in the loop body itself. The same efficiency can be obtained with a less optimally coded example by using compiler optimisations. The hand optimised version is shown to simplify the task of matching the source code against the machine code generated by the compiler.

The example is compiled using a standard RISC compiler. In this example the source was compiled using the ARM software development suite C compiler.

8.3 Region Extraction

The following diagram shows the short sequence of code forming the main loop of the filter:

```
INNER_LOOP:
```

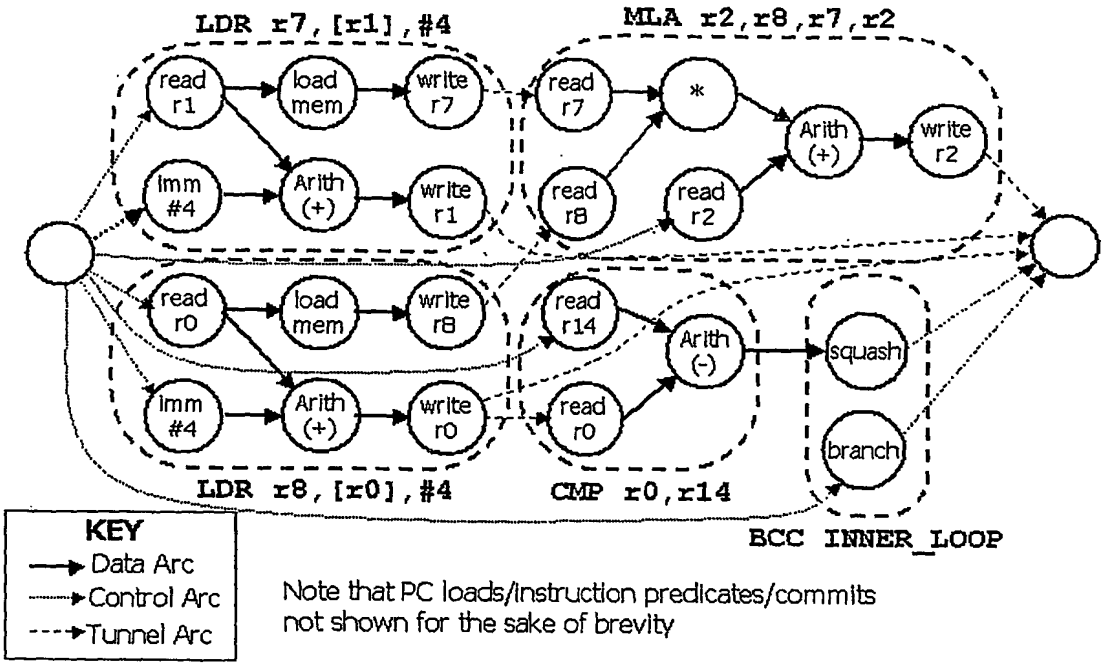
LDR	r7,[r1],#4	Load a sample from memory and increment pointer
LDR	r8,[r0],#4	Load a coefficient from memory and increment pointer
CMP	r0,r14	Check coefficient pointer against the array end – the r14 register is loaded outside the loop
MLA	r2,r8,r7,r2	Multiply sample and coefficient and add result to sum
BCC	INNER_LOOP	Branch back to the start of the loop until all sample/coefficient pairs have been processed

This sequence of ARM instructions will be used as the example code for the rest of the chapter. The code simply loads a sample and a coefficient, multiplies them and adds the result to a cumulative total held in the R2 register. The code makes use of the ARM addressing modes allowing pointers to be automatically incremented. The code also uses the ARM multiply and accumulate operation that combines the multiply and the addition. Comparing one of the pointers against the address after the end of the address range performs the loop end detection.

The control flow analysis identifies the branch to the INNER_LOOP has a backward branch. This allows the sequence of code to be processed as a loop and, as such, the code is allocated to a separate region.

8.4 Code Translation

The diagram below shows the translation of the ARM code into a CDFG representing the base operations to be performed by a CriticalBlue processor:



A dashed line bound the operations generated for each individual instruction. The associated instruction is shown alongside. The CDFG is generated directly from the ARM code without any other intermediate representation.

The load instructions generate six independent operations due to the use of the complex addressing mode. The base register is read, the access performed and the loaded data written to the appropriate destination register. An immediate value of 4 is loaded and added to the base register and the base register updated in the register file in order to perform the address increment.

The MLA (multiply and accumulate) is also translated into five individual operations. The source registers are read and the multiply performed. The accumulator register is then read, the multiply results added to it and then the new total written back to the register.

The compare instruction is translated into three operations. The two operands are read from the register file and a subtraction performed. The same execution unit (arithmetic) is used for both addition and subtraction but with a different method selection. Note that in a real CriticalBlue processor a special adder only unit is used for addressing addition: In this example, the same unit performs the two functions. This is for the sake of simplicity.

The branch is implemented by a squash and branch operation. The squash operation reads the carry output from the arithmetic unit performing the comparison. The squash is used to control the execution of the branch operation, which will be placed in a different strand. The branch operation does not require an immediate operand as the branch is to

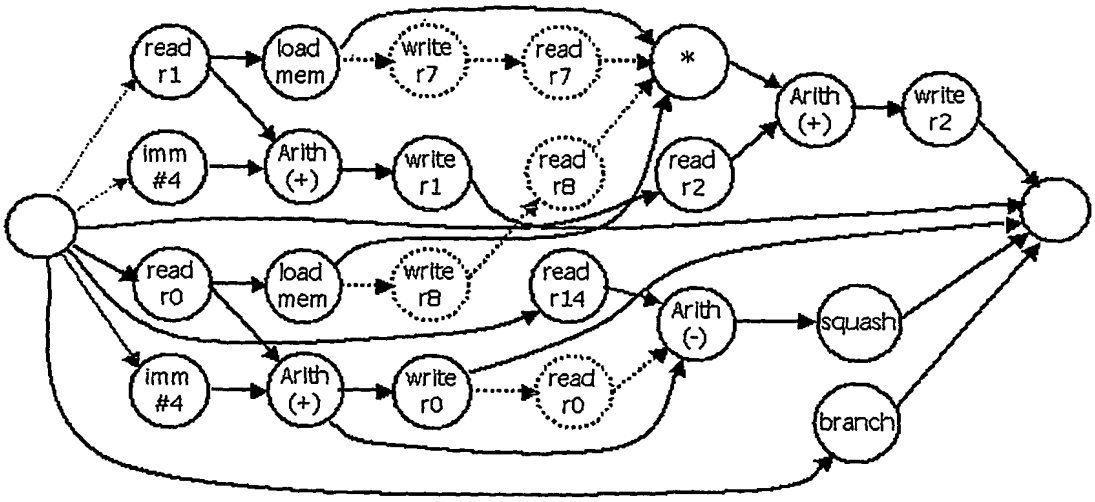
the start of the current region. That address can be simply selected by using a particular branch method. Commit operations for different strands are not shown in this example for the sake of simplicity.

Note that this example does not show PC load, PC register update or instruction predicate check operations. These are normally generated during the translation but deleted from the main code. They are not shown in order to simplify the example.

The translated code will produce the same register results on an ARM instruction granularity basis.

8.5 Transport Optimisation

Once the CDFG for the loop has been built it is optimised in order to remove unnecessary operations. The optimisations applied to the example CDFG are shown below:



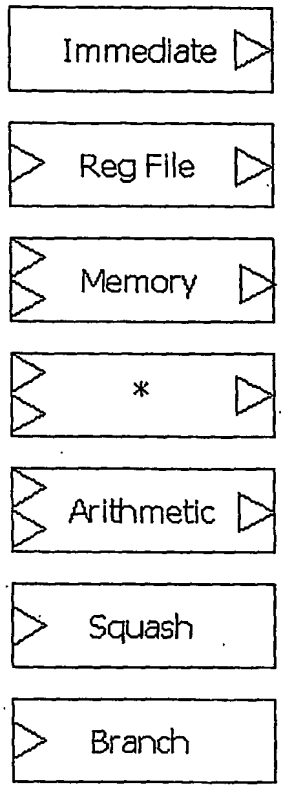
The optimisations are focused on improving the transport of particular data items between operations. This is done by eliminating register reads and writes so that data items can flow directly between operations without go through the register file, wherever possible.

Three applications of register bypassing are applied. The deleted operations and data flows arcs are shown dotted. New data flow arcs are shown that obtain the data from an earlier producer.

Firstly, the write and read of register R7 (one of the parameters to the multiply) can be optimised away as R7 is not live at the end of the loop. Thus data can flow directly from the memory unit to the multiply unit. Secondly, the other operand in R8 to the multiply is similarly optimised. Thirdly, the read of R0 to obtain the current address for comparison can be optimised away. In this case the corresponding write of R0 is retained as the register is live as a loop carried register.

8.6 Minimum Set of Execution Units

The following diagram shows the set of execution units that are required to execute the main loop of the filter:

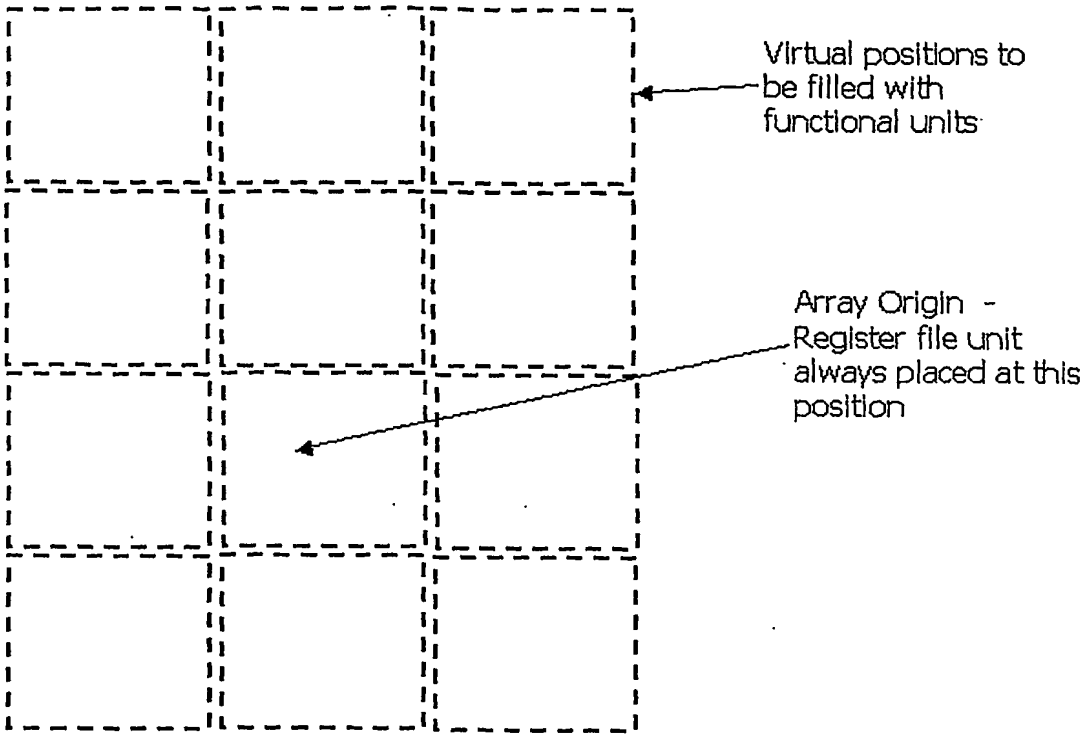


In a complete implementation some other execution units are required (such as a logical unit) to enable the translation of all instructions in the ARM instruction set. However, for the sake of brevity only those units required for the example are shown.

A commit unit is normally required to mark the transition of strands from their speculative to committed execution phases. Again, for the sake of brevity, this is not shown.

8.7 Unallocated Virtual Array

The architectural synthesis must allocate the functional units into an array configuration. Before the allocation commences there is effectively an infinite virtual array into which the units may be allocated. This is illustrated in the diagram below:



The origin position at the centre of the array is always allocated with the register file unit. This unit is always placed centrally because it is likely to require the high levels of communication with the other functional units. Placing it centrally minimises its average distance to all other functional units in the system.

As additional functional units are allocated to the array they are placed adjacently to existing units. The placement ensures that a largely rectangular block of units is generated. The aspect ratio of the allocated blocks can be controlled by the user as an input parameter.

8.8 Data Flow Matrix

The unit placement in the virtual array seeks to place units that communicate frequently close to one another. This means that there is a higher likelihood that they can be connected using direct physical connections. Neighbour units can be allocated direct connections. If the units have to be communicate indirectly then if they are physically close then that minimises the number of intermediate steps required to communicate operands. This reduces the latency of operations.

To direct the unit placement process a matrix is built up representing the data flow within the application. An example matrix is shown below:

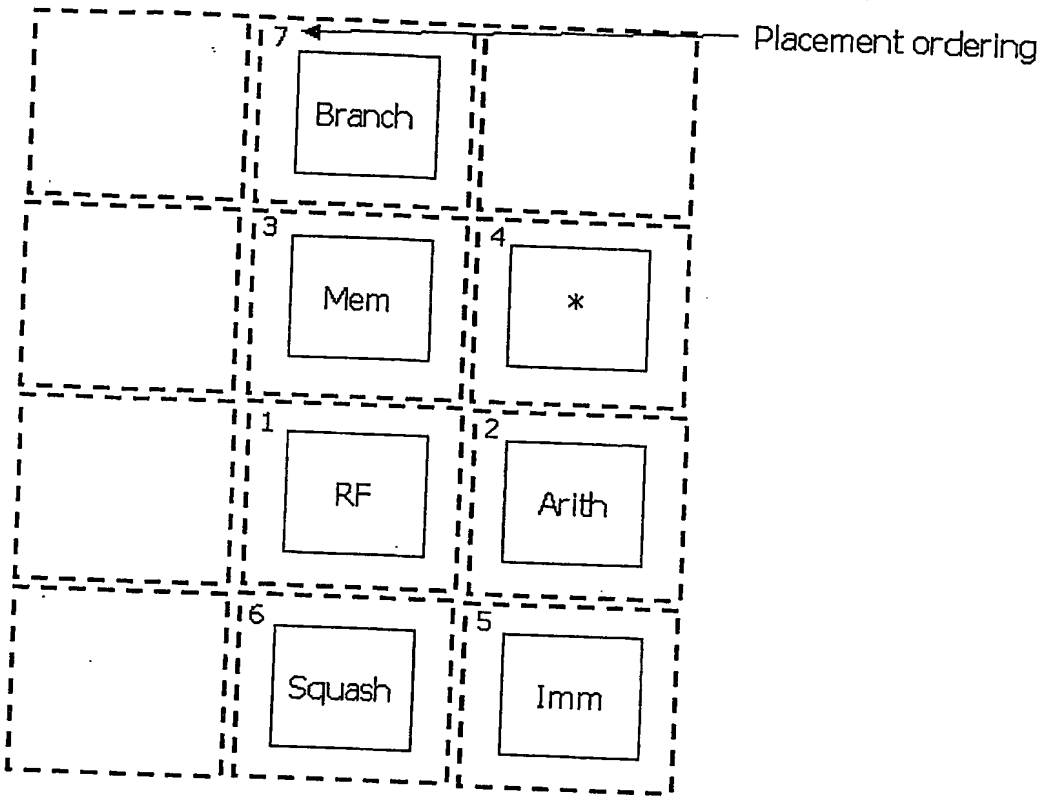
	Arithmetic	Immediate	Memory	Multiply	Reg File
Arithmetic (Op1)	0	0	1	1	3
Arithmetic (Op2)	0	2	1	0	1
Branch	0	0	0	0	0
Memory	0	0	0	0	2
Multiply (Op1)	0	0	0	0	0
Multiply (Op2)	0	0	0	0	0
Reg File	4	0	0	0	0
Squash	1	0	0	0	0

The rows of the matrix represent the input operands of each of the functional units in the system. The columns represent the output ports of the functional units. The values in the matrix indicate the number of transfers that are made from a particular output port to a particular input operand. Thus the quantity of data flow between them is represented. The amount of data flow is also weighted by the relative priority of the functions in which the communications occur. Thus important loop kernels have a much higher impact on the values in the matrix than infrequently executed control code.

The values in the matrix are obtained from an idealised representation of the CDFG. In that representation all possible transport optimisations are performed, independently of whether that could lead to a potential deadlock situation if real code was scheduled from it. The matrix values represent the transfer of data between actual operations rather than with the register file, in so far as possible.

8.9 Unit Placement

Once the data flow matrix has been calculated the next step is to form the placements of the individual functional units. The units are placed so that units with high levels of inter-communication are placed as close to one another as is possible. An example is shown in the diagram below:



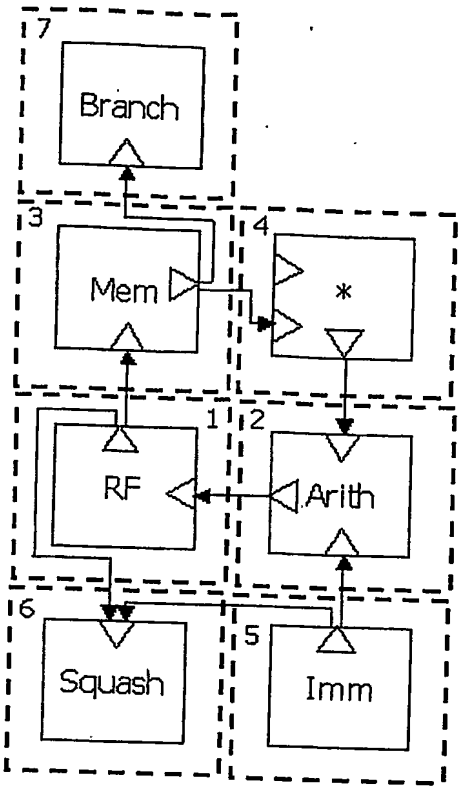
As discussed previously the register file is always placed first. From that point further placements are made by selecting units in reducing order of their frequency of usage. Thus units that are used very frequently are placed near the centre of the array. This allows improved communication with other important units such as the register file. Less frequently used units or those that have been forced to the periphery because they perform large quantities of input/output are placed later. Placements are always made adjacently to previously placed units. Thus in effect the placements are made in a spiral like manner starting with the centrally placed register file.

After the register file unit is added the arithmetic unit may be placed next due to its high level of usage in the example. It is placed adjacently to the register file. The memory unit may then be placed due to its high usage in the loop. Placement continues until all units have been allocated to the array.

8.10 Neighbour Connections

Once all the individual functional units have been placed connections must be generated between them so that they can communicate. There are rules governing the creation of connections in the system. They can only be made between adjacent or otherwise physically close units. This prevents connections being synthesised that are long and thus take up significant area or introduce significant delays to the timing of the system.

The diagram below shows the initial neighbour connections that have been made in the example:



A connection is made between all North, South, East or West neighbours in the array. This ensures that the array structure is communicated effectively to the downstream place and route tools that define the physical layout of the hardware. The spatial model assumed during the architectural creation process is thus communicated to those tools. The direction of the connections between neighbours and ports which are chosen to make the connections are selected on the basis of the most frequently data flows. Again those data flows are determined from the data flow matrix that has been previously constructed.

For instance, the register file input is fed from the output of the arithmetic unit since that is the most frequent source of data that needs to be written to the register file. The output of the register file is fed to the address input of the memory unit. Again, this is because it is frequent data flow. In the case of the register file unit it has only one input and one output but it has three neighbours. Thus one of the input operands or output ports has to be utilised twice in order to generate the required connections. In this case the output connection is also made to the input of the squash unit. This allocation of connections between neighbours continues until all required connections have been made.

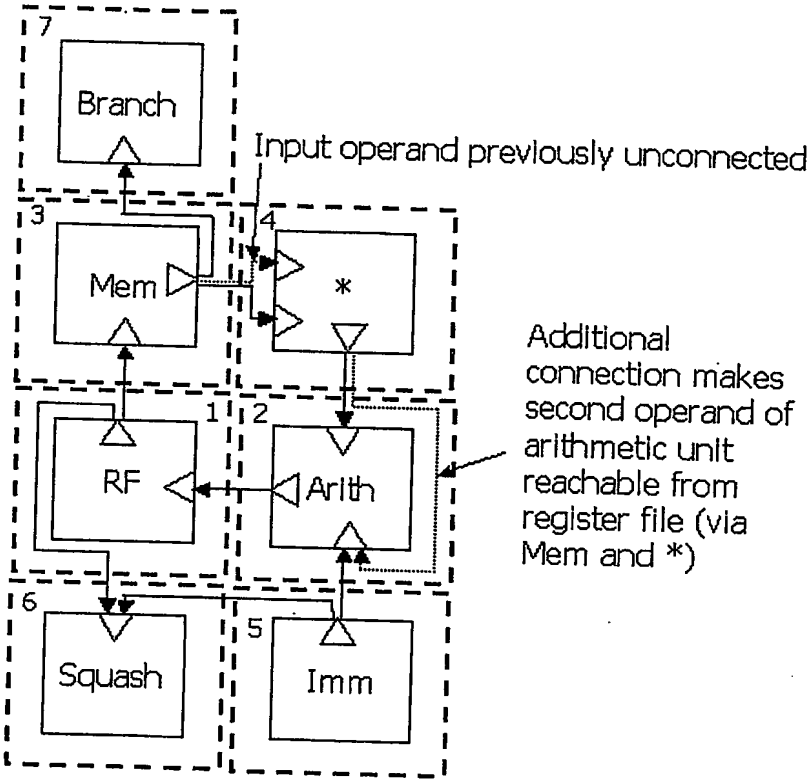
Both the orginal placement of the units and the choice of initial neighbour connections is based on inter-unit communication. Thus many of the connections created should represent some of the key data flows with the application.

8.11 Reachability Analysis

Once the initial connections have been made the network of communication paths are examined and additional connections added as required to ensure reachability. Reachability means that a result from any port of any functional unit can be fed to any operand port of any other functional unit. Thus code can be mapped to any combination of functional units in the system and data can be passed between them as required for

any possible sequence. Obviously most of the paths will be highly indirect requiring the data operands to be copied via functional units a number of times before the final destination is reached. Thus the efficiency of the code is highly dependent upon the exact functional units chosen and the actual data flows required. The placement of units and connections created between them is focused on making those paths efficient.

The additional connections that need to be added for the reachability closure in the example are shown on the diagram below:

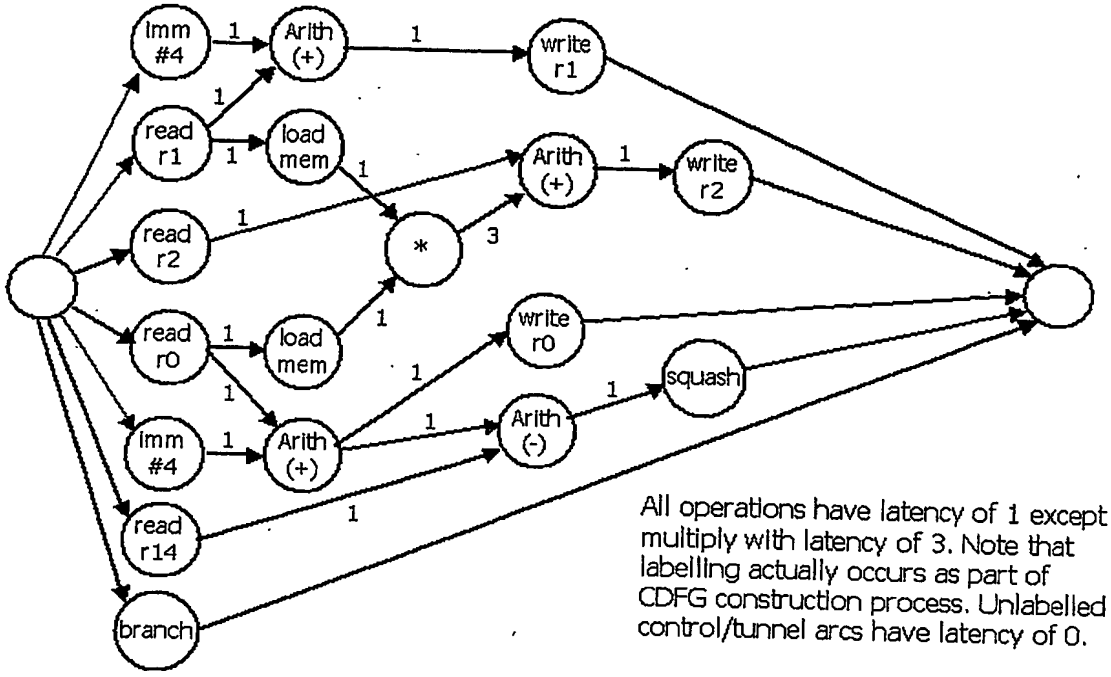


The first operand of the multiplier unit was not previously connected to any result port and is thus not reachable from anywhere. A new connection is added to connect it to the output of the memory unit like the other operand to the multiplier. The second operand to the arithmetic unit was previously only fed from the immediate unit. Since that unit has no inputs it provides no access to any other units within the system. Thus an additional connection must be added to the output of the multiplier unit. This allows results to be fed from all the other result ports in the system.

Once the reachability analysis is complete the array of functional units is fully connected such that data can be passed throughout the array. Additional constraints may also be imposed such that the number of intermediate copies to pass a particular data item from one result port to a particular operand is limited.

8.12 Data Flow Labelling

The next stage is to label the arcs in the CDFG with the latency of the operations. This is illustrated on the diagram below:



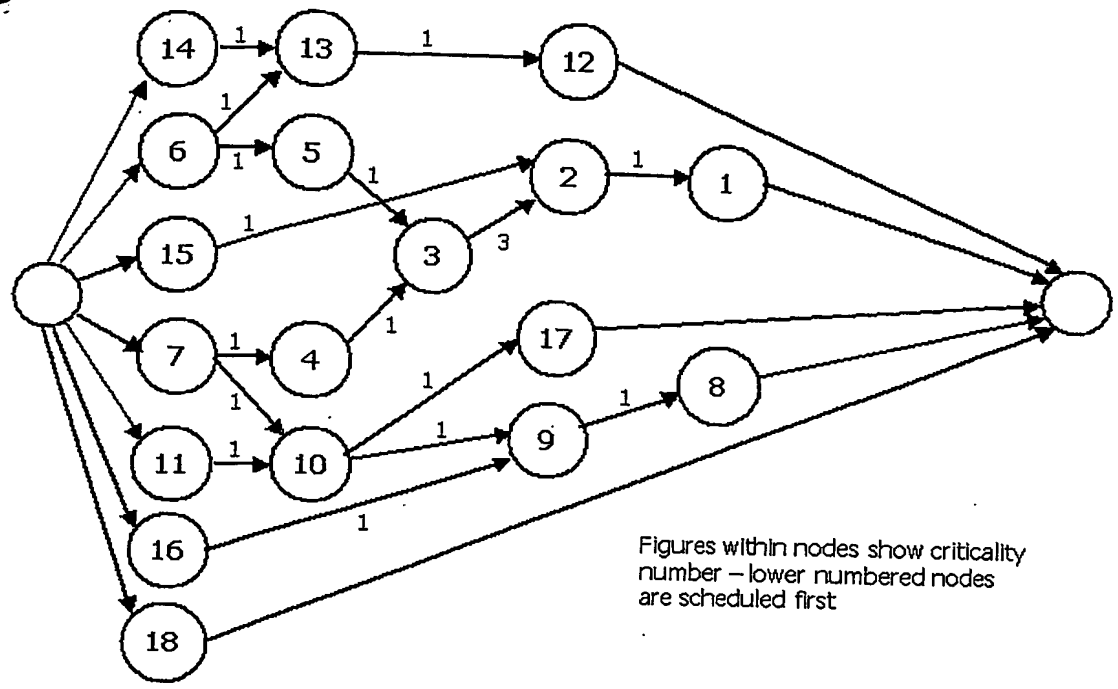
All operations have latency of 1 except multiply with latency of 3. Note that labelling actually occurs as part of CDFG construction process. Unlabelled control/tunnel arcs have latency of 0.

In this case all operations have a latency of 1, except for the multiply which has a latency of 3. The arc labeling actually occurs during the construction of the CDFG as the latencies are fixed properties of the execution units required by each operation. The labeling is shown here as a separate step for the purposes of the example.

The cost labeling is used during the node criticality analysis to determine the most critical nodes in the CDFG. These are the nodes that are in critical paths of data flow and whose placement in the schedule is likely to have the most impact on overall execution time.

8.13 Node Criticality Analysis

Once the arc labeling is completed, node criticality analysis is performed. Possible results of this for the example are shown on the diagram below:

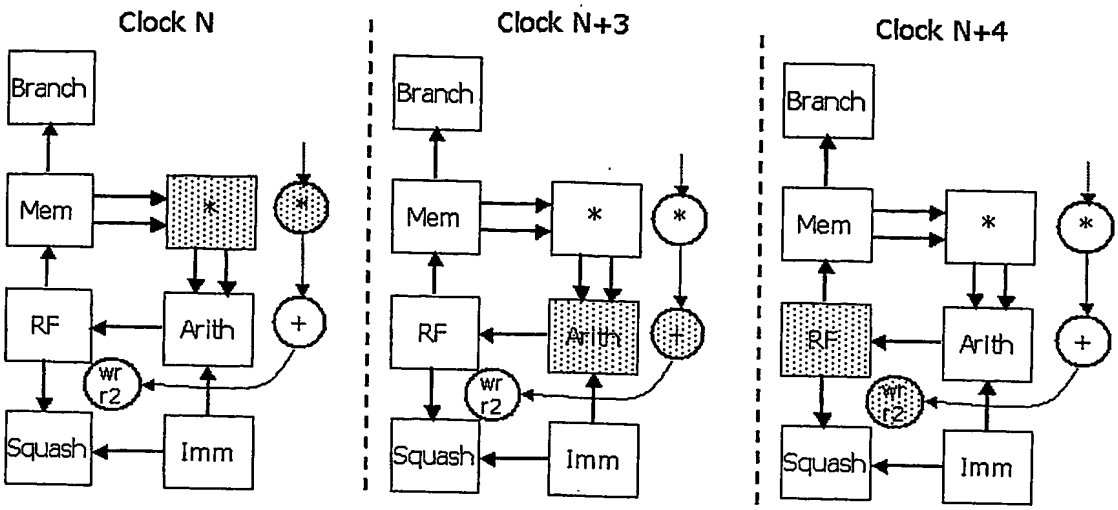


Each node is marked with a number indicating its criticality. Lower numbers are more critical than higher numbers. When performing scheduling, nodes with lower numbers are scheduled before nodes with higher numbers. This allows more critical nodes to be scheduled earlier and thus getting a better choice of placement in the code. A node can only be scheduled when all its dependent nodes have already been scheduled so these provide a further limitation in scheduling order.

The numbers are derived from the node free float calculated as part of the critical path analysis. The free float is a measure of how much the placement of node in the schedule can be varied without affecting the overall execution time of the region. Highly critical nodes on the critical data flow path have a free float of 0.

8.14 Mapping to Architecture

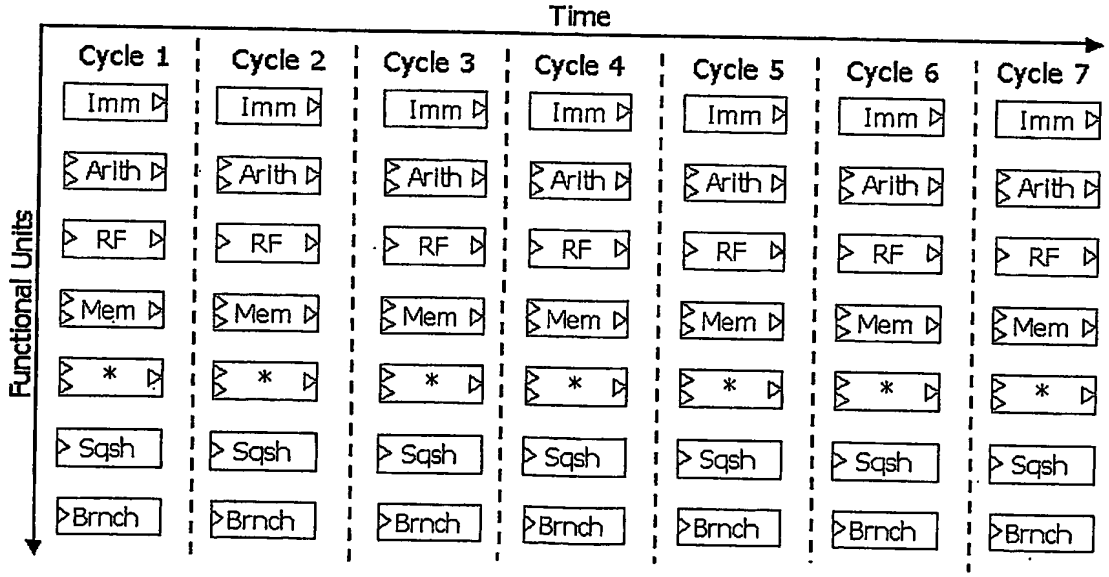
The diagram below illustrates the scheduling problem of mapping the CDFG to the architecture of the processor. A small segment of the CDFG is shown:



At clock N the multiply operation is mapped onto the multiply unit. As the multiply has a latency of 3 clock cycles the next operation cannot be initiated until clock cycle N+3. At this point the add operation is performed on the arithmetic unit. Since data can be fed directly from the multiplier to the arithmetic unit no intermediate copy operations are required and the two calculations can be executed back-to-back. Finally at clock cycle N+4 the register write is performed on the register file unit. Again the units are directly connected so no additional copy operations are required.

8.15 Unit-Time Representation

The diagram below shows the initial Unit-Time representation for scheduling on the example architecture:

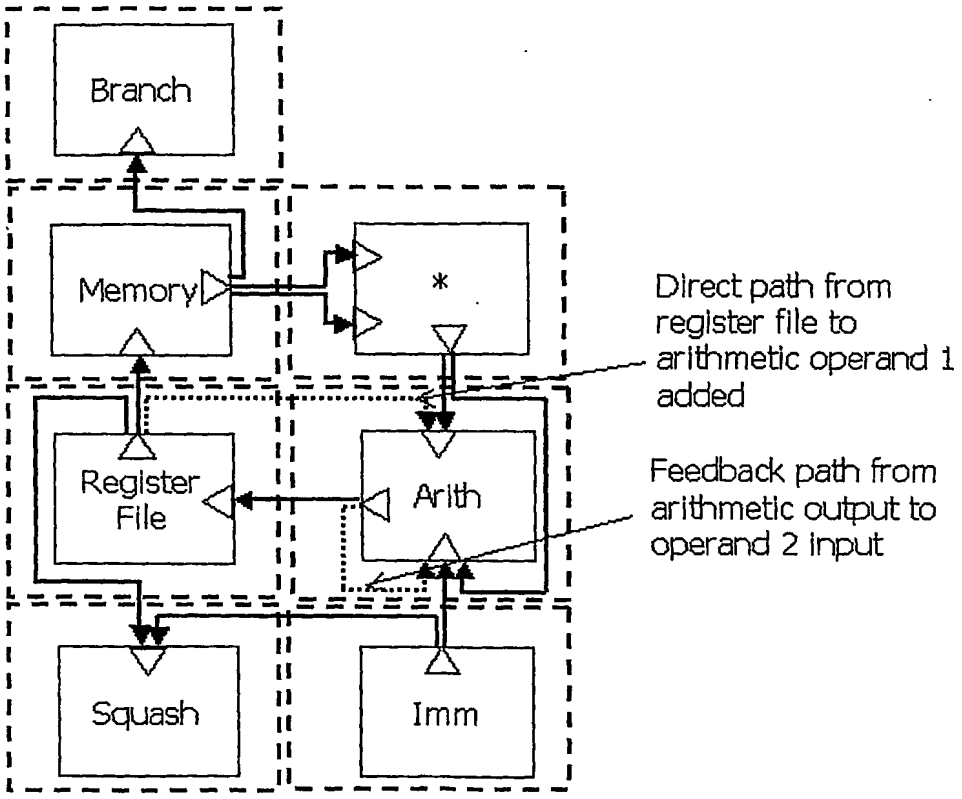


On each clock cycle there is an entry for each functional unit in the processor. The table is used to show a graphical representation of how those units are used over time. The purpose of the scheduling process is to minimise the overall schedule length (i.e. total number of clock cycles to execute a given block of code) by making as much use of parallelism between functional units as possible.

8.16 Connectivity Optimisation

If the filter loop code is marked as being critical then it is used to customize the connectivity of the final architecture. Additional connections are added that correspond to data flows in the loop. Addition of these connections improves the performance that can be obtained. Connections are added one by one. That is, a single connection is added and then code is scheduled again to measure the connections impact on the schedule generated and determine what other connections could be added.

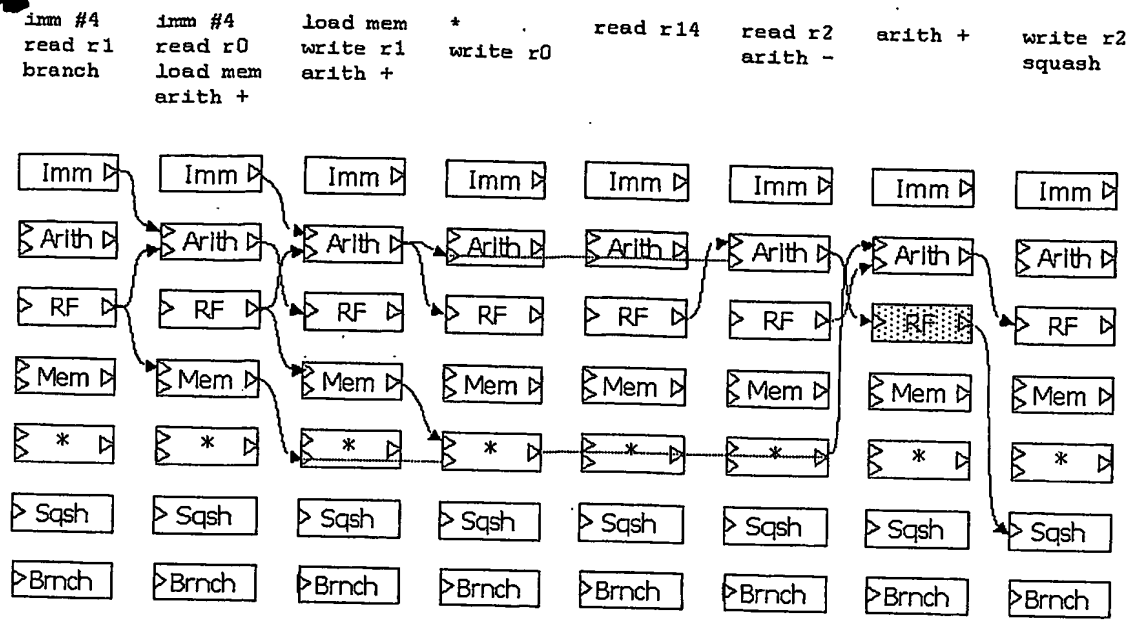
The first two custom connections that may be added by the example code are shown in the diagram below:



Firstly, a direct connection is made between the register file output and the first operand of the arithmetic unit. This allows the address registers to be read directly into the arithmetic unit in order for the addresses to be updated. A second direct connection is added between the output of the arithmetic unit back into its second operand. This allows cumulative results to be calculated without having to perform copy operations through other functional units. In effect this forms the accumulate section of a multiply-accumulate unit. The connections required for a multiply-accumulate unit emerge automatically in the architecture just from analysis of the filter loop dataflow.

8.17 Final Schedule

The impact of the custom connections on the schedule are shown in the unit-time diagram below:



The legends above the blocks of functional units show the set of operations being executed on each clock cycle. As illustrated a number of operations are completed on each clock cycle. The configuration of connections between the execution units (as configured by the measured data flows) ensures that the majority of data flows are directly from one unit to the next. In most cases the data arrives on exactly the clock cycle that it is required so that it is used immediately. One exception is the second operand to the multiplier where the data is available one clock cycle earlier than when it needs to be used. The multiply operation cannot start until the second operand is available. The data is simply held in the appropriate output register from the memory unit until it is required. Note that the result of the multiply is not read for three cycles after the operation starts as the multiply unit has a latency of that many clock cycles.

The shaded operation executed on the register file indicates a copy operation. The copy allows data to be passed from the arithmetic unit to the squash unit as these are not directly connected.

The register file unit is accessed on every single clock cycle. If more parallelism was available in the loop then the average number of register file accesses would tend to reduce, thus relieving pressure on that unit.

The customized data paths allow the entire loop to fit into 8 clock cycles. Higher average performance is possible with a loop containing more parallelism. However, such a loop would be too complex for a worked example.

Note that the issue of the squash on the last cycle of the schedule would not be legal on a real CriticalBlue architecture, as it would have to precede the commit for the strand holding the branch operation. These commit operations are not shown in this example, however.

8.18 Summary

This example has illustrated how a specialised architecture is automatically generated using the data flow in example code to direct the connectivity. The architecture is optimised for execution of a FIR loop. However, many of the connections that have been added to the architecture will be commonly used by other, unrelated, code and provide

significant performance benefits. A direct connection is automatically generated between the output of the arithmetic unit and one of its inputs to form an accumulate unit. The arithmetic unit is also placed next to the multiply unit so that results can flow directly into one of its operands. In effect, a custom multiply-accumulate unit has emerged from the optimisation.

The example used consists of only five host instructions. Therefore the amount of parallelism available is minimal. With a larger example much more parallelism is available and thus there is more scope to use many more of the functional units simultaneously.

More parallelism can be obtained from the FIR filter by unrolling the loop by hand so that each iteration performs a number of multiply-additions. The CDFG optimisation will be able to eliminate more register file accesses as a higher proportion will be to registers that are not live outside of the loop and are not loop carried. With full optimisation selected, the compiler may also perform loop unrolling.

The MetaMapper is itself capable of performing loop unrolling. This was not shown for this example for the sake of brevity. However, the example loop would be unrolled to increase the number of operations in the loop and thus potential parallelism. MetaMapper unrolls loops where there is a potential of an exit on each original iteration by mapping multiple iterations to different strands in the same region. Once unrolled a multiply could be initiated on every other cycle. The performance bottleneck is the need to obtain two data items from memory for each calculation. The code could be rewritten to model another a second memory unit. The operands can then be obtained from different memory units and allow a performance of one multiply-accumulate per cycle to be achieved. With further memory units added the performance can be grown even further.

9 Structural HDL Creation

9.1 Synthesis Methodology

A structural HDL description of the processor is generated from the description file. This HDL description instantiates all the functional units in the array and provides the modelled connectivity between them. The HDL indicates the layout of the processor array so that the place and route stages are directed such that the spatial model used during architectural creation correlates with the actual layout.

9.2 Testability

The full application code can be tested and debugged using the simulation framework supported by CriticalBlue. However, this is only a software simulation using the behavioural models supplied by the user. It does not directly test the hardware of the processor. Any processor hardware produced by the CriticalBlue tools must be testable, both prior to the commitment to silicon and in a manufacturing flow. There are two essential aspects to this testing. They are unit level and system level testing.

For unit level testing each of the individual control and functional blocks in the processor need to be tested. Test benches with appropriate stimulus and expected responses must be produced. Test benches and vectors will be supplied for the fixed control units that are developed by CriticalBlue and are present in each processor.

System level testing involves testing the entire processor including all the interconnections between different control and functional units within it. The interconnections between the various functional units are generated automatically so should not be incorrect. Before producing silicon, hardware simulations need to be performed to verify that is the case.

For the manufacturing test flow they do need to be tested since they might be damaged by a silicon defect.

The CriticalBlue tools produce the entire processor as a soft core in RTL. Existing RTL simulation tools can be used to simulate the entire system running application code produced by the CriticalBlue compilers. As the design is progressed through logic synthesis and the place and route stages, further simulations are performed at the gate level to provide confidence in the design. The application code used would normally include test benches for the individual execution units in order to verify their functionality.

To assist with manufacturing test, the CriticalBlue tools are able to automatically generate test vectors to verify the interconnection network between the functional units. The vectors also test the logic generated automatically for each functional unit. These vectors are concerned with testing the parts of the design that are produced automatically by the CriticalBlue tools.

The test vectors use the debug chain is used to inject particular data patterns into the output registers of the functional units. The vectors test all the data paths in the design and the operation of the multiplexers. A special test bypass is provided to route data from the input operands of a functional unit to its output without using the execution unit. This allows the functional unit connectivity to be tested in isolation from the execution unit functionality. The test vectors are different for every design because the connectivity is different. Complete coverage of all the automatically generated parts of the design is guaranteed. This allows the engineer to concentrate on verification and testing efforts on the parts of the design that they have produced.

The testing of the individual execution units is the responsibility of the user, as it would be for any other hardware design block. The CriticalBlue tools cannot generate vectors to test them since it doesn't know their internal structure. The engineer must produce a series of stimulus and expected outputs for the execution unit, usually generated from a test bench. These are designed to test the full functionality and gates/connectivity within the execution unit. Although the CriticalBlue tools cannot actually test the execution units, they provide support by generating test vectors that present the correct stimulus and reads the outputs from a particular execution unit.

9.3 Dedicated Input Ports

9.3.1 RSN Port (hw_RSN)

The RSN port may be routed to particular execution units. The Result Strand Number (RSN) shows the strand number of any new operation being initiated on the functional unit. The RSN is obtained from an immediate value specified for an operation.

For instance, memory units containing an SWB use `hw_RSN` to determine which strand a particular memory operation is associated with. This allows a speculative write to be committed or discarded as required depending upon the final status of strands in the region.

The simulation environment provides `hw_RSN` has a global integer variable. The behavioural model of an execution unit may use this. It is set to the appropriate value before calling the behavioural model of each execution unit.

9.3.2 SEM Port (hw_SEM)

The SEM port may be routed to particular execution units. The Strand Enable Mask (SEM) shows which strands are currently being executed. There is one bit for each

possible strand in the processor. If a particular strand bit is reset then that indicates that the strand is disabled and no operations from the strand should be initiated.

For instance, memory units containing an SWB use `hw_SEM` to determine which entries to discard at the end of the region. Entries in the SWB from enabled strands are committed while those from disabled strands are discarded.

The simulation environment provides `hw_SEM` has a global integer variable. The behavioural models of execution units may read it.

9.3.3 ERF Flag (`hw_ERF`)

The End Region Flag (ERF) may be routed to particular execution units. The ERF is a single bit from the execution word that indicates that the word is the last in the current region. ERF is used by execution units that need to perform particular actions at the end of a region.

For instance, memory units containing a SWB use `hw_ERF`. The assertion of the end flag indicates that speculative writes have become committed and that writes from squashed strands should be purged.

The simulation environment provides `hw_ERF` has a global Boolean variable that may be read by behavioural models of execution units.

9.4 Dedicated Output Ports

9.4.1 Wait Flag (`hw_wait_x`)

This is a dedicated wait flag that may be asserted by any execution unit. If asserted then the entire pipeline of the processor is stalled. A number of individual wait ports are available to allow several functional units to have a wait capability. The *x* value gives the wait signal number.

The wait must be asserted very early in the last execution cycle of the functional unit. It must be available early as it is combined with all other wait signals and routed to all functional units. When wait is asserted the clocks for all functional units except those asserting wait is gated off. Thus the same state is maintained in the other functional units. If clock gating is not available then the assertion of the wait signal causes register feedback in each of the pipeline stages of the functional unit. Thus the pipeline is not advanced while wait is asserted.

The wait flag is used for functional units that may occasionally have an extended latency. The frequent shorter latency is specified as the fixed latency of the unit and then the wait flag is used to extend the latency when required. Possible uses of this mechanism are as follows:

- Implementation of a cache. The wait is asserted when there is a cache miss. While the processor pipeline is stalled the cache unit can access main memory to obtain the required cache line. A minimum latency of two clock cycles is generally required to allow sufficient time for a tag lookup, comparison and then assertion of the wait signal early in the second cycle of execution. A one cycle latency cache could be implemented but at the expense of an elongated cycle time.

- Implementation of denormalised floating point operations. Denormalised values (if implemented) do not occur frequently but usually require an extra cycle of processing for floating point operations. Assertion of the wait line allows an extra cycle to be inserted.

- If a Speculative Write Buffer (SWB) is being used on memory units then the wait signal can be used to delay the pipeline if the buffer becomes full. The pipeline is stalled while entries are written to memory and free positions become available.

Note that if the wait signal is used then the maximum interrupt latency (for both fast and slow interrupts) is extended by the maximum possible wait period.

For simulation purposes the variable `hw_wait_x` is declared as global boolean. The behavioural model for an execution unit may write to the variable in order to stall the simulation pipeline. This allows cycle accurate simulation results to be obtained even when pipeline stalls are present in the system.

9.4.2 Squash Vector (`hw_squash_x`)

A squash vector is generated by a squash operation and used to reset bits in the Strand Predicate Mask (SPM). A number of individual squash vectors are supported in order to allow the use of multiple squash units. This allows parallel squash operations to be performed if a single squash unit is a performance bottleneck. The number of the squash bus is indicate by the value *x*.

Each squash vector is a set of bits with one bit for each possible strand in the processor. If the bit is asserted then the corresponding bit in the SPM is cleared. This disables the corresponding strand and prevents any results from the strand being committed. All squashes for a strand must be resolved before it enters its committed phase.

The branch control unit generates one squash vector. This allows it to squash all strands higher than a strand that is performing a branch. These later strands must be squashed as they are logically unreached if an earlier branch is taken. The squash ensures that those strands are not committed.

An individual squash operation is able to generate several squash flags. The squash flags are set by the code generation process and allow control flow constructs to be collapsed into strands. A squash is predicated on a particular conditional value supplied to the squash unit.

Note that the squash unit must ensure that the bits in the squash vector are reset for cycles in which no valid squash operation was issued.

For simulation purposes the variable `hw_squash_x` is declared as a global integer. This may be written with appropriate squash bit vectors by the behavioural model of a squash unit. The simulation kernel reads these variables in order to update the simulated SEM value showing which strands should be executed.

9.4.3 Abort Bus (`hw_abort_x`)

An abort bus is generated by a check hazard or guard operation and used to reset bits in the Strand Abort Mask (SAM). A number of individual abort buses are supported in order to allow the use of multiple check hazard units. This allows parallel check hazards to be performed if a single unit is a performance bottleneck. The number of the abort bus is indicate by the value *x*.

Each abort bus specifies a particular strand number with a separate valid bit to indicate if the strand number is valid. This causes the bit in the SAM corresponding to the strand to be cleared. This aborts the strand.

Note that the check hazard or guard unit must ensure that the valid bit of the abort bus is reset for cycles in which no valid operation was issued.

For simulation purposes the variable `hw_abort_x` is declared as a global integer. This may be written with the appropriate abort strand by the behavioural model of a unit. The simulation kernel reads these variables in order to update the simulated SAM value showing which strands should be aborted.

9.4.4 Commit Bus (`hw_commit_x`)

A commit bus is generated by a commit unit and used to updated the Committed Strand Mask (CSM). This in turn enables the selection of more possible branch destinations from the branch control unit. A number of individual commit buses are supported in order to allow the use of multiple commit units. The number of the commit bus is indicate by the value *x*.

Each commit bus specifies a particular strand number with a separate valid bit to indicate if the strand number is valid. This causes the bit in the CSM corresponding to the strand to be set if that strand is currently enabled. This allows any branch issued by the strand to be selected if no previous branch has been issued. If a breakpoint has set on the strand then this causes the selection of the breakpoint branch.

Note that the commit unit must ensure that the valid bit in the commit bus is reset for cycles in which no valid operation was issued.

For simulation purposes the variable `hw_commit_x` is declared as a global integer. This may be written with the appropriate commit strand by the behavioural model of a unit. The simulation kernel reads these variables in order to update the simulated CSM value showing which strands have been committed.